CDRL: B017
25 March 1994

UNISYS

Component Provider's and Tool Developer's Handbook

Central Archive for Reusable Defense Software (CARDS)

Informal Technical Data

CARDS

Central Archive for Reusable Defense Software

STARS-VC-B017/001/00
25 March 1994

CDRL: B017
25 March 1994

INFORMAL TECHNICAL REPORT For The SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Component Provider's and Tool Developer's Handbook
Central Archive for Reusable Defense Software
(CARDS)

STARS-VC-B017/001/00
25 March 1994

Data Type: Informal Technical Data
Contract NO. F19628-93-C-0130
Line Item 0002AB

Prepared for:

Electronic Systems Center
Air Force Material Command, USAF
Hanscom AFB, MA 01731-2816

Prepared by:

Unisys, Valley Forge Engineering Center
and
EWA, Inc.
under contract to
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

CDRL: B017
25 March 1994

INFORMAL TECHNICAL REPORT For The SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Component Provider's and Tool Developer's Handbook
Central Archive for Reusable Defense Software
(CARDS)

STARS-VC-B017/001/00
25 March 1994

Data Type: Informal Technical Data

Contract NO. F19628–93–C-0130
Line Item 0002AB

Prepared for:

Electronic Systems Center
Air Force Material Command, USAF
Hanscom AFB, MA 01731–2816

Prepared by:

Unisys, Valley Forge Engineering Center
and
EWA, Inc.
under contract to
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

CDRL: B017
25 March 1994

Data Reference: STARS-VC-B017/001/00
INFORMAL TECHNICAL REPORT
Component Provider's and Tool Developer's Handbook
Central Archive for Reusable Defense Software
(CARDS)

CDRL: B017
25 March 1994

Data Reference: STARS-VC-B017/001/00
INFORMAL TECHNICAL REPORT
Component Provider's and Tool Developer's Handbook
Central Archive for Reusable Defense Software
(CARDS)

Principal Author(s):

Roslyn Nilson *Date*

Paul Kogut *Date*

George Jackelen *Date*

Approvals:

System Architect: *Kurt Wallnau* *Date*

Program Manager: *Lorraine Martin* *Date*

(Signatures on File)

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** 25 March 1994	**3. REPORT TYPE AND DATES COVERED** Informal Technical

4. TITLE AND SUBTITLE Component Provider's and Tool Developer's Handbook CARDS	**5. FUNDING NUMBERS** F19628-93-C-0130
6. AUTHOR(S) Roslyn Nilson, Paul Kogut, George Jackelen	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Unisys Corporation, 12010 Sunrise Valley Drive, Reston, VA 22091	**8. PERFORMING ORGANIZATION REPORT NUMBER** STARS-VC-B017/001/00
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of the Air Force, Headquarters Electronic Systems Center, Hanscom AFB, MA 01731-5000	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** B017
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Distribution "A"	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 words)*

See Abstract Page

14. SUBJECT TERMS	**15. NUMBER OF PAGES** 120
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	SAR

CDRL: B017
25 March 1994

Data Reference: STARS-VC-B017/001/00
INFORMAL TECHNICAL REPORT
Component Provider's and Tool Developer's Handbook
Central Archive for Reusable Defense Software
(CARDS)

ABSTRACT

This Component Provider's and Tool Developer's Handbook was developed under the Central Archive for Reusable Defense Software (CARDS) Program to help facilitate software reuse adoption. Government developers and software industry vendors supporting government acquisitions are provided with guidance for developing domain-specific, reusable components and tools supporting reuse. The goal of this handbook is to stimulate the development and commercialization of large-scale components and tools for vertical domains. Focus is placed on architecture-centric, library-assisted software reuse. It is assumed the reader is familiar with the design and development of software. The audience for this handbook consists of:

- government: domain and System Program Office (SPO) engineers,
- contractors: component creators and tool developers.

Table of Contents

1 INTRODUCTION

1.1 Purpose

This Component Provider's and Tool Developer's Handbook (herein handbook) provides government software developers and industry vendors (e.g., government contractors and commercial software creators) with guidance for building/creating domain-specific reusable components and tools to facilitate software reuse. The goal of this handbook is to stimulate the development and commercialization of large-scale reusable components and tools for vertical domains. This handbook complements the information contained in the Central Archive for Reusable Defense Software (CARDS) Program's (herein CARDS) *Engineer's Handbook* [CARDSc], which discusses how the government can monitor the work, processes, and products described in this handbook. The end users of these components and tools are government programs system developers. The components and tools may be accessible to multiple audiences through government software repositories (e.g., CARDS, Asset Source for Software Engineering Technology (ASSET) Program (herein ASSET), and Defense Software Repository System (DSRS)).

An infrastructure of reusable components and tools is critical to the success of software reuse. This handbook describes how to develop domain-specific components for reuse by other applications within a domain and gives guidance for developing reuse-related tools. Emphasis is placed on domain-specific, architecture-centric, library-assisted reuse. The methods and processes described herein can apply to domains, other reuse libraries, and other reuse programs (library-assisted or not). However, they are based on experiences and lessons learned from the cooperative work on CARDS and other (e.g., Portable, Reusable, Integrated Software Modules - PRISM) programs.

This is an updated version of [CARDSp].

To understand these concepts, methods and processes, the reader should be familiar with software development in general.

1.2 Scope

This handbook is intended to increase awareness of envisioned changes in government and vendor activities resulting from Department of Defense (DoD) adoption of domain-specific reuse techniques. This handbook provides a range of information on reuse topics such as domain analysis, component creation, and tool development.

1.3 Audience

The main audience for this handbook is government domain engineers, and component and tool creators (either government or contractors) (see Figure 1-1). However, government System

Program Office (SPO) engineers/managers should also be familiar with the contents of this handbook because they use the components and tools (application engineering) and they are often involved in the development of components and tools.

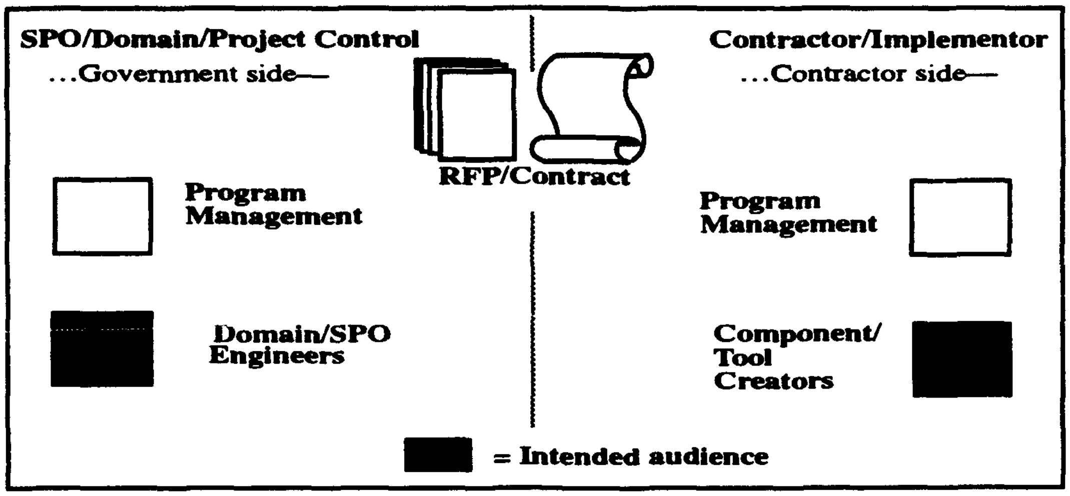

Figure 1-1 Intended Audience

Domain engineers are primarily concerned with creation of a component and tool base which can be managed and used within a domain. In this context, component and tool creation consist of [CARDSc]:

1. domain analysis (defining the 'problem space'),
2. domain design (proposing a 'solution space'),
3. domain implementation (implementing the proposed 'solution space').

"Domain engineering activities can be viewed as analogous to application engineering activities (see Figure 1-2). However, it is important to note that domain engineering and application engineering are two distinct processes. The products of domain engineering activities can be (and are meant to be) utilized in an application engineering activity which may provide feedback to influence future domain engineering activities. The focus of application engineering is a single system (e.g., F-22 avionics), whereas the focus of domain engineering is on multiple related systems (e.g., avionics for all fly-by-wire aircraft) within a domain." [CARDSc]

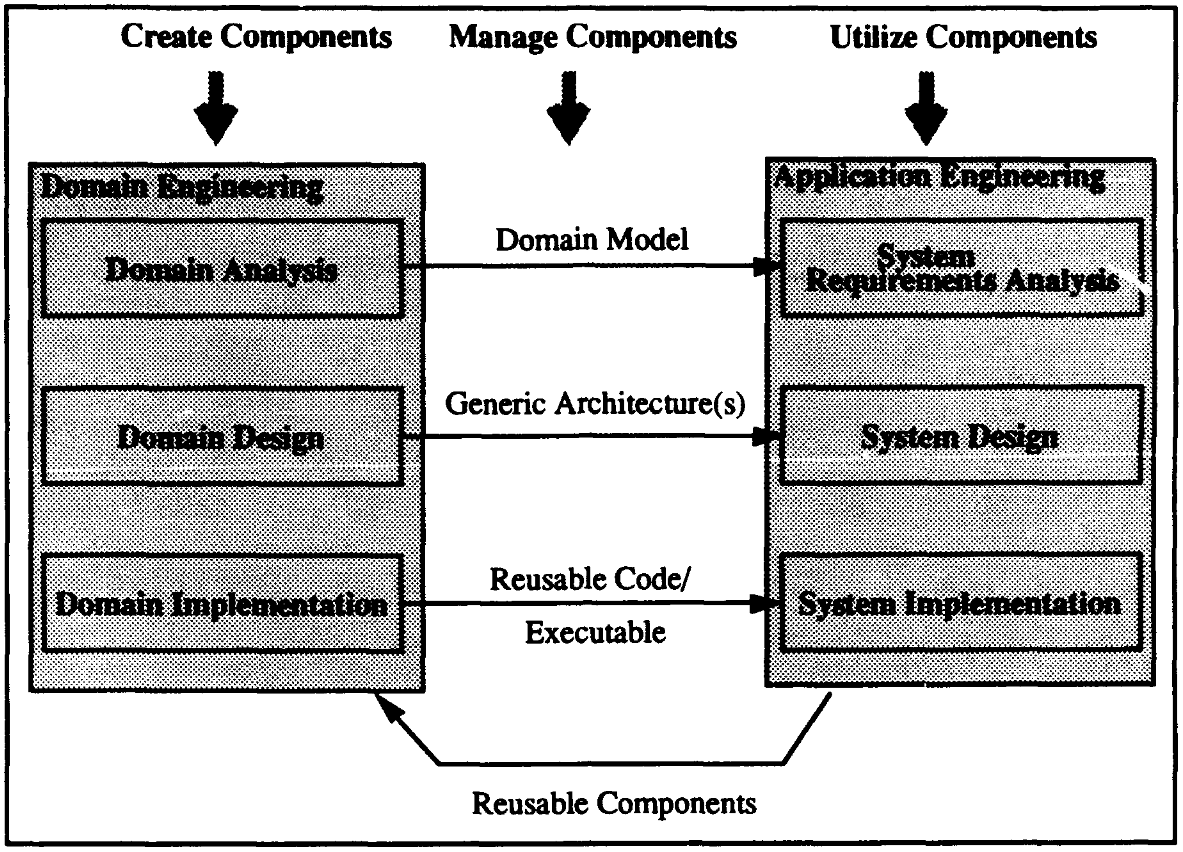

Figure 1-2 Domain and Application Engineering

For this handbook:

- Component and tool creators develop reusable products and/or aids (tools).
- SPO engineers are responsible for:
  - application engineering,
  - delegated domain engineering activities.

When engineering activities are contracted for, domain and SPO engineers are involved in the following activities:

- Requests for Proposal (RFP),
- proposal evaluation,
- monitoring of engineering activities after contract award,

- monitoring of ongoing sustaining/maintenance engineering efforts of fielded products.

1.4 Terminology

The following lists some basic terminology used in this handbook [NIST93] (see Appendix C for a more complete list):

- **Architecture**: Often used as a synonym for a design. However, the term usually refers to a specification documenting the way in which pieces are integrated to form a whole, as in the components of a software system.
- **Component**: Any unit that can potentially be reused.
- **Domain**: A distinct functional area that can be supported by a class of software systems with similar requirements and capabilities.
- **Domain analysis**: Systems analysis applied across a class of domain software systems.
- **Domain architecture**: A high-level structure or design for domain software systems.
- **Domain engineering**: The process of defining the scope (i.e., domain definition), specifying the structure (i.e., domain architecture), and building components (e.g., requirements, designs, software code, and documentation).
- **Domain management**: Typically the management of a group of similar system development and maintenance efforts, and conducts domain analysis and design to increase productivity and reduce costs. Also provides appropriate access to domain-relevant information on all applications to the domain analysis and design teams [DISA93a].
- **Generic architecture**: The high-level design and constraints characterizing the commonalities and variances of interactions and relationships among system components.
- **Reuse**: The adaptation of a reusable component to more than one software system.
- **Reuse library**: A collection of reusable components/assets, together with the procedures and support functions required to provide the reusable components.
- **Tools**: Items contributing to the reuse infrastructure within an organization and can be applied to automated reuse processes, e.g., creating/maintaining/modifying and storing/retrieving reusable components.

1.5 Background on Reuse

CARDS (concerted DoD effort to transition advances in the techniques and technologies of domain-specific software reuse into mainstream DoD software procurements) goals are to:

- produce, document, and propagate techniques to enable domain-specific reuse throughout the DoD,
- develop and operate a domain-specific library system and necessary tools,
- develop a franchise plan which provides a blueprint for institutionalizing domain-specific, library-centered reuse throughout the DoD,
- implement the franchise plan which uses and provides a tailored set of services to support reuse.

In addition to CARDS, there are various efforts in the software community currently addressing organizational, business, legal, and technical aspects of software reuse, which have captured and produced a wealth of knowledge to assist domain and SPO engineers:

- The Defense Information Systems Agency (DISA) which has, among other things, begun several process development initiatives in the domain analysis and reuse metrics collection areas. To date, DISA has defined the domains existing within DoD [DISA92a], developed guidelines for conducting domain analyses [VITALETTI], and developed a method for defining and collecting reuse-related metrics [DISA93b].
- The Software Engineering Institute (SEI) has developed a domain analysis process (Feature-Oriented Domain Analysis - FODA) and is developing Model Based Software Engineering (MBSE) [WITHEY93]. This process focuses on the use of models and core components in software development and evolution. The SEI has shown how to combine domain analysis and architectural concepts in the context of MBSE [PETERSON93].
- NATO (North Atlantic Treaty Organization) Communications and Information Systems is supporting the reuse of software life-cycle products in NATO contracts. They have developed guidance for managing reusable libraries, developing reusable components and reusing exiting components [NATOa][NATOb][NATOc].
- The Advanced Research Projects Agency (ARPA) sponsored Software Technology for Adaptable, Reliable Systems (STARS) Program (herein STARS) is increasing software productivity, reliability, and quality by integrating software processes and reuse concepts. Specifically, STARS reuse concepts provide a conceptual foundation, framework, and requirements for reuse technology processes and supporting

tools. Their approach is generic with respect to organizations, software engineering methodologies, technologies, and environments. [STARS90]

- ARPA's Domain Specific Software Architecture (DSSA) Program is doing research on domain modeling, architecture representations, and tools/environments/processes for reuse and megaprogramming.
- ASSET, a STARS effort, focuses on technologies to permit reuse libraries to effectively interoperate and develop a library for software assets/components and a national clearinghouse for software reuse sources. ASSET, CARDS, and DSRS have implemented reuse interoperation among the three libraries [ASSET92].

In addition to documentation currently available, these efforts continue to provide engineers and others with the latest advances in the rapidly changing and emerging field of software reuse.

1.6 Assumptions

It is important to note that the focus of this handbook is on domain engineering and component creator duties. The government or contractor may create components and/or tools. This handbook assumes government personnel (domain and SPO engineers) oversee/manage the effort or results.

Discussions within this handbook assume domain-specific (i.e., applies to only one domain), architecture-driven (i.e., based on a given architecture), library-assisted (i.e., components can be found in a reuse library) reuse.

It is assumed that domain engineering (for an application domain being considered) is part of the reuse process and that component development and reuse are architecture driven. In addition to the above assumptions, this handbook assumes some form of domain management office is responsible for the domain engineering activities (i.e., responsible for managing components within a domain [KATZ93]). Reuse of high-level components (e.g., domain architectures) and large-grained code components is emphasized.

To fully utilize the concepts in this handbook, it is recommended that domain and SPO engineers be familiar with software development techniques and methodologies, existing government regulations (such as *DoD-STD-2167A*, *MIL-STD-499*, and *MIL-STD-1521B*), and the acquisition process. Appendix G outlines additional consultation sources.

The roles in creating reuse tools are more ill-defined and flexible than the roles for creating domain components. However, the approaches used (i.e., the processes used in Chapters 3 and 4) are similar. Domain engineers in the domain management office are responsible for developing/acquiring a set of reuse tools. This may be done as part of a broader tool standardization/acquisition effort (e.g., Computer Assisted Software Engineering (CASE) tools):

- If an required reuse tool is domain independent, then domain engineers can work with a reuse technology provider to develop/acquire the tool.

- If the required reuse tool is domain-specific (e.g., application generator), then the tool would be developed/acquired like a normal reusable component with the involvement of the domain engineer, component/tool creator, and SPO engineer.

1.7 Document Overview

This handbook has been organized as follows:

A. Chapter 2, Reuse Overview, is an overview of software reuse and domain-specific reuse in particular. Chapter 2 can be skipped by those familiar with software and domain-specific reuse.

B. Chapter 3, Software Component Development, explains the component development process, starting with domain analysis and modeling, through requirement analysis and component design/code to component testing. Specific engineering tasks and responsibilities are described.

C. Chapter 4, Reuse Tool Development, examines methods for identifying and evaluating existing tools; discusses key issues in defining requirements for tools; and highlights design conventions, procedures for testing, and architectures. The emphasis is on reusable tools becoming part of a reuse library (thus, there may be overlapping tasks and responsibilities between the tool creators and the reuse library maintainers). These reuse tools can be used for component creation, management, and utilization. Tool acquisition may involve evaluation and selection of existing tools or building new tools.

   Chapters 3 and 4 are organized by life cycle phases and provide examples and methodologies for each phase.

   As stated before, the government can/does do development; but this handbook assumes most of the development is done by a contractor.

D. Appendix A, References, lists the references for this handbook.

E. Appendix B, Acronyms/Abbreviations, lists the acronyms and abbreviations used in this handbook.

F. Appendix C, Glossary, contains a glossary of terms used in this handbook.

G. Appendix D, Developing Reusable Component, provides guidelines for creating reusable source code.

H. Appendix E, Sample Reuse Tool Descriptions, describes some available reuse tools.

I. Appendix F, Sources of Reusable Components, lists some sources for reusable components.

J. Appendix G, Related Documents, provides a list of CARDS and non-CARDS products and services, plus a brief description of each.

2 REUSE OVERVIEW

2.1 Software Reuse Background

Software reuse is a process in which software resources are applied to more than one system. It can occur across similar systems or in widely different systems [DoD92]. All resources or components resulting from the various stages of the software development process are potentially reusable. These components include: domain model, software architecture, product design, and implementation components (source code, test plans, procedures and results, and system/software documentation) [CARDSa].

Software reuse can be classified in several different ways. Reuse can be practiced:

- **Ad-hoc**, when there are no defined methods for performing reuse.
- *Opportunistic*, when it is up to software developers to identify when reuse is possible, to locate reusable components and to integrate them.
- **Systematic**, when there exist defined procedures for leveraging existing components on future software projects. Efforts are devoted up-front to create a suitable process. Knowledge of how and when to reuse software components within a domain is incorporated into the process [WARTIK92].

Early reuse efforts focused on small-grained general-purpose code components [BIGGER89] (e.g., subroutines, object libraries, or Ada packages). Developing systems reusing only small-gained code components do not in most cases provide as much cost savings as does reuse with large-grained components (e.g., database or geographical information system). Reuse of high-level components (e.g., requirements, architecture, design and test documents) along with large-grained code components provide an even greater economic benefit. The systematic reuse of high-level components is beneficial because reuse is introduced early in the life cycle, paving the way for additional reuse. For example, if an existing set of requirements is reused it is likely the associated architecture, and many designs, code, tests and documentation components can also be reused. The potential benefits of a systematic software reuse process employing high-level components can be great. These benefits can include increased productivity, increased system quality and reliability, and reduced maintenance costs.

Establishing a process for any activity requires an initial investment and long-term commitment. There are many costs associated with transitioning from an old process to a new one. These costs include efforts to identify domains and customers for components through market studies. Ensuring legal issues are addressed is another cost associated with implementing reuse. Developing a systematic reuse process requires up-front investment and commitment. The investment can be large. However, once the investment has been made, the long-term savings can be great. The CARDS *Direction Level Handbook* can be referenced for information on reuse experiences resulting in benefits and quantitative productivity increases [CARDSb].

2.2 Domain-Specific Reuse

The reuse of high-level components and large-grained code components is usually more feasible in the context of a domain. A domain is an area of activity or knowledge containing applications sharing a set of common capabilities and data. Domains can be classified as either vertical or horizontal (see Figure 2-1) [ARMY92]. A vertical domain is a class of system such as command and control or weapon systems. A horizontal domain is another class of systems; however, it is also a subsystem of many other larger domains. For example, the user interface domain is a subdomain of the command and control domain and the information systems domain [CARDSa].

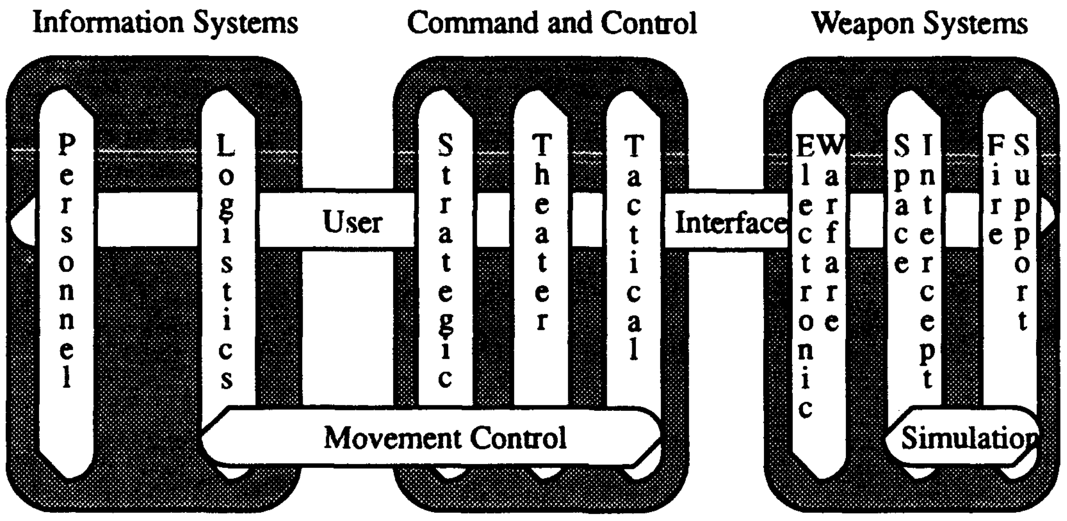

Figure 2-1 Vertical and Horizontal Domains

Domain-specific reuse is the reuse of components in a specific domain to build an instance of an application in that domain and can result in greater savings than general-purpose reuse. As a domain matures, so matures the body of knowledge about it and experience in it. The number of existing systems and domain experts, from which information can be drawn, increases. Components maintained and refined as the domain matures become more reliable and effective [STARS92].

The process supporting domain-specific reuse is domain engineering (see Figure 2-2), which begins with domain analysis. Domain engineers develop a domain model and generic architecture for the domain analyzed, and create reusable components based on the model and generic architecture. Domain engineering results in the organization of domain knowledge for future development use. Application engineers can then use domain engineering products to develop new applications.

Domain analysis involves identifying, collecting, organizing, analyzing, and representing the relevant information in a domain. This is based on the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within the domain (see Figure 2-3) [VITALETTI92]. Domain analysis products include a domain dictionary, context diagrams, and domain models [DISA92a]. Domain analysis is described in more detail in Section 3.2.

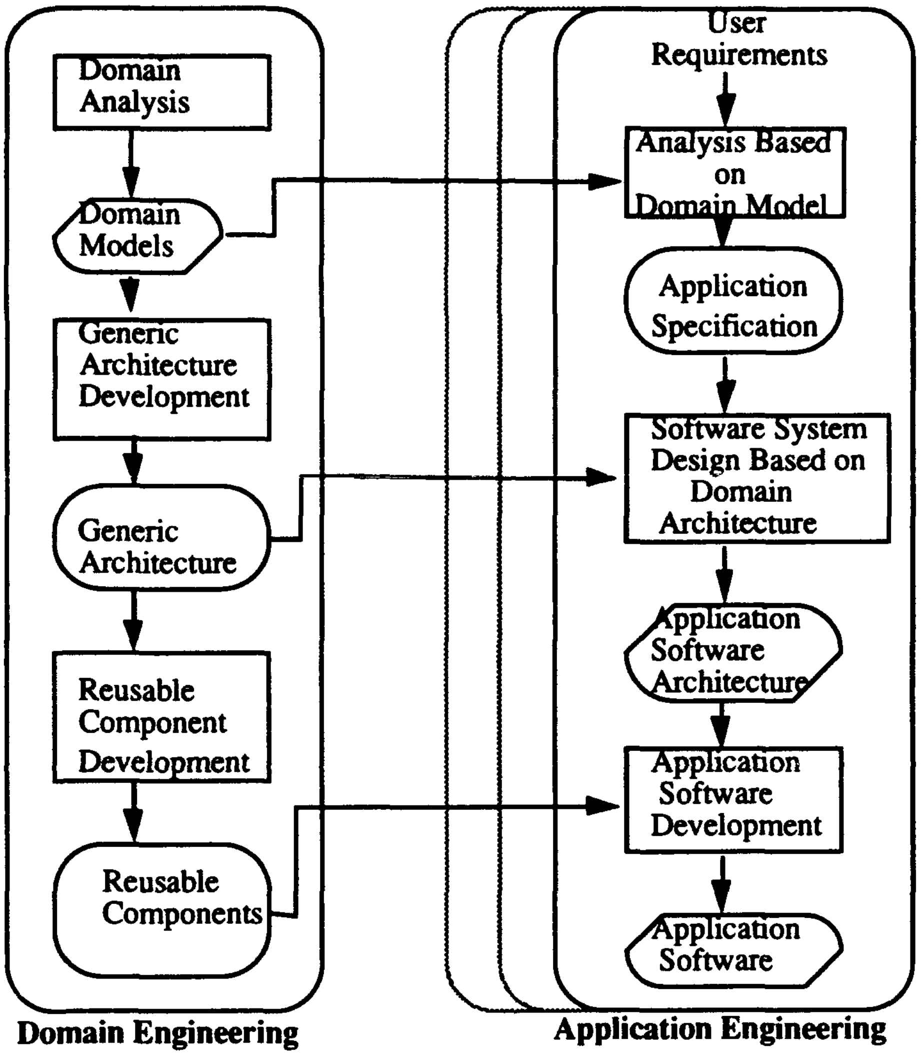
Domain Analysis
Domain Models
Generic Architecture Development
Generic Architecture
Reusable Component Development
Reusable Components
Domain Engineering
User Requirements
Analysis Based on Domain Model
Application Specification
Software System Design Based on Domain Architecture
Application Software Architecture
Application Software Development
Application Software
Application Engineering

Figure 2-2 Domain Engineering

Using the domain knowledge gathered during domain analysis, domain engineers develop a generic architecture. Using the generic architecture, domain engineers acquire and, where necessary, create reusable components catalogued into a reuse library for use by application engineers. This activity is architecture-driven since the development of the components is based on the generic architecture. The goal of an architecture-driven process is to achieve "black box" reuse [DoD92]. If one views a system or subsystem as a black box meeting certain requirements (e.g., a COTS product), and if these systems are reused in building other systems, we have potentially reduced the full cost of developing a system/subsystem to satisfy the reuse requirements. This concept of "megaprogramming" is strongly espoused by the DoD, and is already being implemented in the acquisition of some systems.

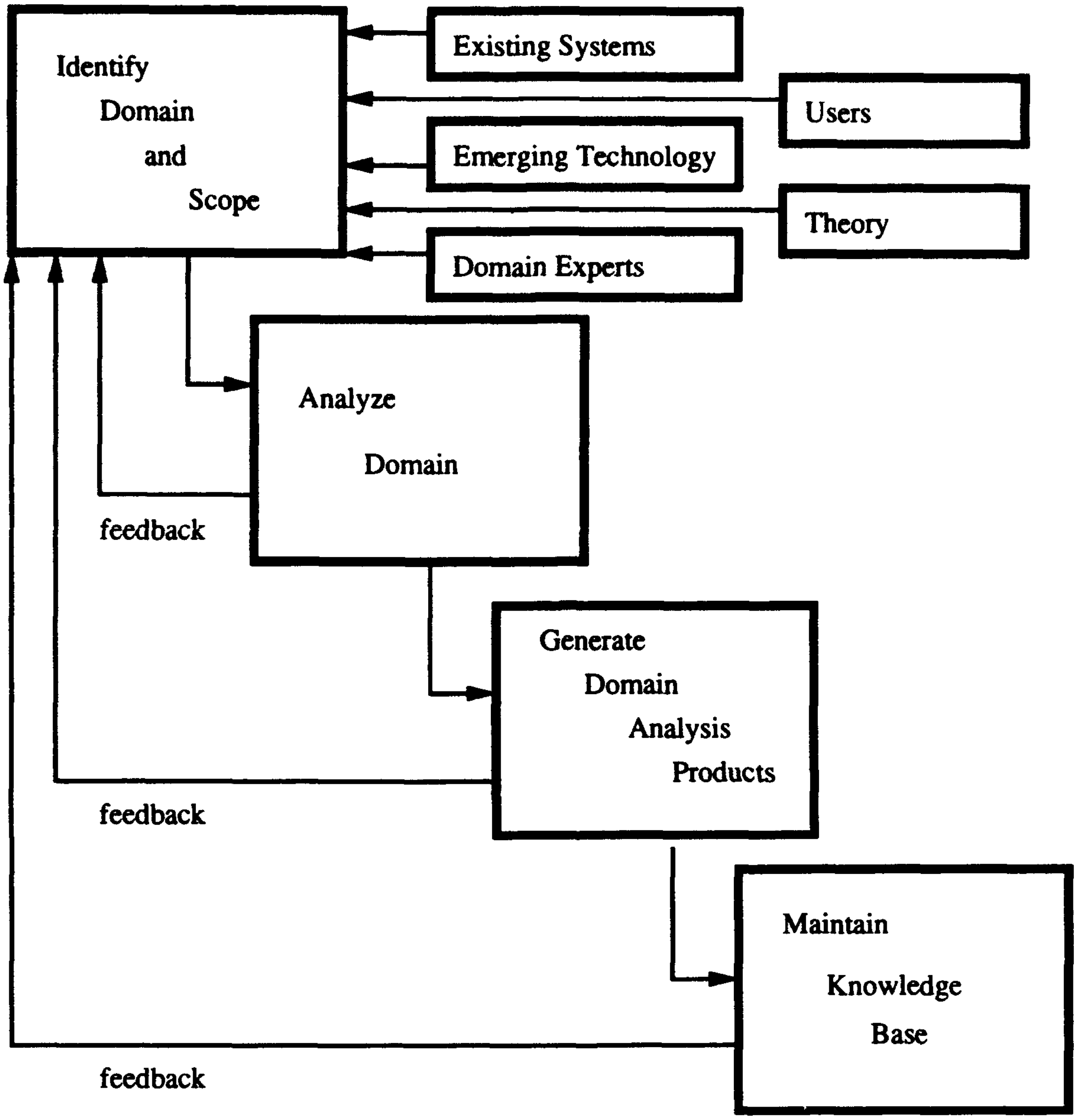

Figure 2-3 Domain Analysis

The *DoD Software Reuse Vision and Strategy* document [DoD92] defines the concept of centrally managed reuse within a domain. CARDS has recommended this be done via a domain management office responsible for organizing and maintaining domain knowledge [CARDSa][CARDSb]. A domain management office typically spans across several program offices and will be staffed by government domain engineers. These domain engineers will perform the domain engineering tasks described above, with support from SPO engineers and government or contractor component creators. The specific assignment of domain engineering tasks will depend on the existing organization and the available resources. Domain engineers should have expertise in the present and future "big picture" of the domain (across many systems). SPO engineers should have expertise in specific areas of the domain and in existing domain systems. Component creators should have expertise in developing high quality reusable software and should also have expertise in the domain (since they may be involved in doing the domain analysis). Domain engineers are responsible for monitoring and evaluating the development of the domain model, generic architecture, and components; but they will often delegate some of this effort to SPO engineers (especially component development, because components often come from ongoing application engineering or reengineering efforts). SPO engineers will also serve as the focal point for application engineering [CARDSc]. All three roles (domain engineer, SPO engineer, and component creator) must understand the domain engineering and application engineering processes.

2.3 Reuse Libraries

A domain-specific library is an instrument supporting domain-specific reuse. It is not the complete reuse solution. Current domain-specific libraries can be classified as either [WALLNAU92a]:

- **Component-based libraries** are similar to book libraries. They can be thought of as software warehouses. The central focus of the component-based library is the component.
- **Model-based libraries** are organized to reflect the relationships among reusable components. The focus of a model-based library is the domain model and generic architecture.

Regardless of whether the library is component-based or model-based, a typical domain-specific library contains a domain model which may include representations of the generic requirements of the domain, a generic architecture, implementation components, documentation of components, and lessons learned. A model-based library uses a domain model or generic architecture as its infrastructure; contains a mapping between the domain model, generic architecture, and implementation constraints; and may provide tools to facilitate the reuse of these products.

Reuse libraries should have a process for continuous improvement. As with any process improvement program, feedback is essential and must be provided by the application engineering process to the domain engineering process. It is through feedback that components are refined and maintained. As part of the configuration management task, application engineers (who are modifying components, enhancing them, and who are identifying bugs), must pass this information back to the domain engineer. Providing this feedback requires additional effort; however, it is only through feedback that component quality and reliability can be increased.

A component residing in a domain-specific, architecture-based reuse library usually has undergone certain basic qualification and certification procedures (see Section 3.6.3 for details). The component may have been qualified based on domain requirements, architectural constraints (relations/interfaces among components), and implementation constraints (constraints to be satisfied when components are composed into specific systems). It also may have been certified for reliability, maintainability, portability, and security [CARDSd]. Certification criteria can include [DoD92]:

- Requirements have been validated in an operational system.
- Component documentation exists, facilitating modifications and maintenance.
- Code meets established standards.
- There is documented evidence of successful, perhaps frequent, reuse.
- *The component is warranted by some organization.*
- Components are submitted with test plans, procedures, and results.

2.4 Reuse Tools

Domain-specific reuse leverages large amounts of problem space and solution space knowledge. Since tools are needed to capture and use this knowledge, tools are used at various stages of domain engineering. Often the same tool is used in a different way during application engineering. For example, a reuse library usually has some form of searching/browsing tool. A library may also contain tools for composition of components and code generation. Other tools help develop and evaluate requirements, domain models, architectures, code components, and tests for a domain. Reuse tools include some standard CASE tools, as well as tools developed explicitly for specific reuse tasks. Some tools lock the user into a specific architecture style (e.g., object-oriented), whereas others are more flexible. Reuse tools (see Chapter 4) are part of a complete software engineering environment; therefore, the issue of tool integration is important.

2.5 Reuse Processes

Now that some basic concepts have been introduced, it is possible to discuss a general framework for domain-specific reuse processes to help organize and focus this handbook. A domain-specific, model-based reuse process can be divided into three distinct subprocesses [STARS92]: Asset Creation, Asset Management, and Asset Utilization (see Figure 2-4). Note that STARS uses the term "asset" *whereas* CARDS uses *the* term "component."

- The asset creation process produces and evolves domain assets such as: domain models, domain architectures, application generators, software assets, etc.

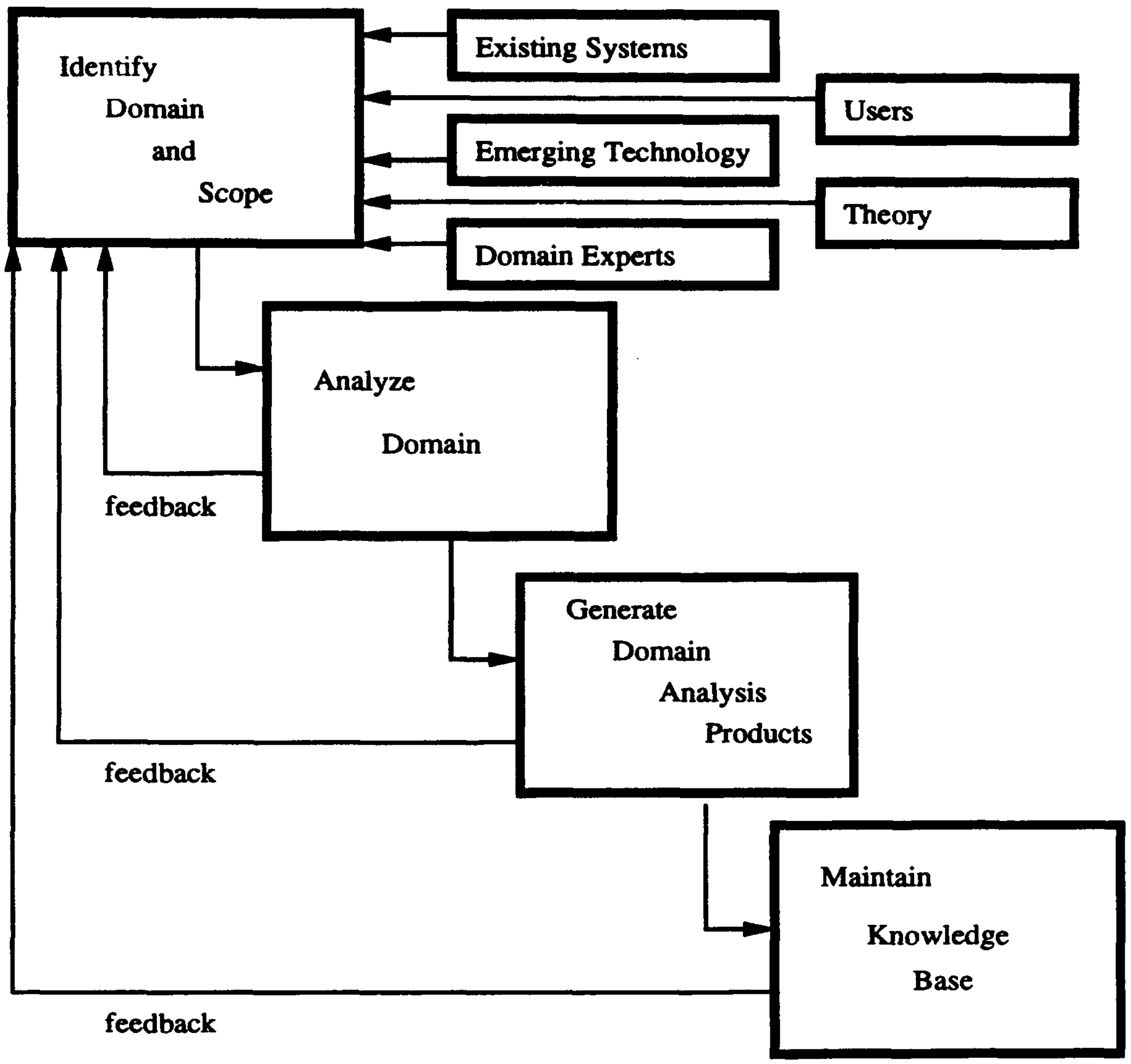

Figure 2-4 Reuse Process

Organizations chartered to create an asset base for a specific domain enact the asset creation process. For example, PRISM is chartered to create assets for the Air Force command and control domain [CARDSe].

- The asset management process acquires, evaluates, and organizes assets produced by the asset creation process. Asset management acts as a brokering mechanism between the asset creators and asset utilizers.

The CARDS Command Center Library (CCL) is an example of a library acting as a broker in enacting the asset management process. The CCL manages PRISM generated assets by providing the requisite services for the acquisition, evaluation, and organization of assets. Normally, any process embodying data management and evolution functions of domain information can be classified as an asset management process [CARDSe].

- The asset utilization process uses assets from the asset management process (and produced by the asset creation process) to identify, select, and tailor desired assets and integrate them to create application systems within the target domain.

  Any organization chartered with the acquisition of a new system in the target domain is an example of an asset utilization agent.

It is conceivable that a single organization may enact more than one of the above processes. However, a clear delineation is necessary to support the concept of multiple stakeholders in a domain: producer, broker, and consumer. These roles may be assigned to different DoD organizations.

2.6 Market Studies of Software Reuse

Identifying opportunities for implementing reuse is important. Devising a market study that canvases product lines within a particular service or agency should provide a good overview of need within an organization. However, this study should also identify those persons with responsibility for making purchase decisions, so that they may be appropriately targeted by marketing executives, using data resulting from the market study to bolster one's presentations. Control over DoD funding differs markedly from that present within the commercial sector, and must be thoroughly investigated and appropriately addressed to ensure success in future marketing approaches. For example, technical personnel at development and maintenance levels may be convinced of the need for particular components and tools; however, if government executives are not similarly convinced, or lack sufficient information to make an informed decision, funding to purchase such new technology will be nonexistent.

Familiarizing oneself with available DoD, service and/or agency policy and mission statements (e.g., *DoD Software Reuse Vision and Strategy* [DoD92], the *Army Strategic Software Reuse Plan* [ARMY92] and its companion *Software Reuse Implementation Plan* [ARMY93a], the *Air Force Software Reuse Implementation Plan* [AF92], and the *Navy's Reuse Implementation Plan* [NAVY93]) should help determine which direction to pursue in implementing market studies. Furthermore, determining compatibility and identifying links between one's line of business, product lines and DoD demonstrated need should better focus efforts. One possible source of such information is the Army's Communications-Electronics Command (CECOM) *Advanced Planning Briefing for Industry* [ARMY93b], which provides the general thrust of CECOM requirements for software acquisition under their "Best Value" approach.

One complicating factor is the existing state of flux within the government regarding its domains, and the uncertainty of funding for specific product lines, increasing the difficulties associated with predictive market study results. At the DoD level, these issues are being addressed by placing increased emphasis on the establishment of domain managers, as well as the institution and accomplishment of domain analyses. Once a domain management structure is in place, policy statements will emanate from one central location, thus lessen the effort associated with identifying and querying responsible parties. Additionally, when domains receive sufficient emphasis to guarantee thorough analyses, resulting in fully delineated product lines, there will exist less uncertainty within particular domains, and marketing analyses should be easier to conduct and exhibit increased reliability.

The ultimate intent should be to identify alignment between one's current commercial ventures and established or projected government needs. One means of accomplishing this aim is to canvas existing product line libraries to determine their contents, and ascertain similarities between the focus of such libraries and one's commercial product line, thereby identifying probable areas for future development. A potential source for ideas about how best to approach one's marketing efforts is [BLAN92].

Another factor that should guide the selection of particular product lines is the organizational and technical flexibility to retool or dramatically change the focus of development efforts midstream. Before contemplating and implementing a dramatic change in focus, one must gain an awareness of the magnitude of change required, both in terms of organizational structure and design, as well as development and manufacturing alterations. While the market study will identify potential business areas for concentration, management must determine those aspects of the current business environment requiring change in response to new tool or component development, and appropriately ready the organization for the new conditions. In preparation for such change, management should work closely with technical personnel to determine whether any change in methodology or process should be undertaken, whether existing personnel require additional training or education to successfully handle the change, how anticipated changes should be best communicated to those who must respond, how change should be implemented (e.g., whether in a small, selected project, and then gradually spread throughout the organization, or whether it should be implemented organization-wide at the outset), and who should guide and manage its implementation.

2.7 Legal Issues Relating to Software Reuse

Legal issues impact the development and reuse strategy of software components. In the software reuse community, many perceive that legalities can prohibit software reuse. However, since these legal issues are a subset of management issues, they should be treated within the context of a business strategy [REUSE93a]. Furthermore, following discussions are only a high-level overview of the legal topics. Legal counsel should always be consulted before any decisions having legal ramifications are made.

There are two types of legal issues pertinent to managers of domain engineers, component developers, and SPO engineers. These are rights and responsibilities with respect to intellectual property rights and risk/liability resulting from the distribution and reuse of components.

Intellectual property is an intangible output of rational thought processes having some intellectual or informational value. Intellectual property can be protected by patents, copyright or trade secrets [PERMAR93] (See Appendix C for definitions/explanations of terms). Various levels of rights can be negotiated under government contracts. For software, these are unlimited or restricted. For technical data, these rights can be unlimited, limited and Government Purpose License Rights. (Note: under the Federal Acquisition Regulation (FAR), software documentation is treated as software and has software rights; under the Defense Federal Acquisition Regulation Supplement (DFARS), software documentation is treated as technical data and has data rights, not software rights coverage). These rights determine usage of the software and documentation. However, no matter what rights are negotiated, the contractor maintains "ownership" (i.e., they hold the copyrights).

The main concern regarding intellectual property rights in the component development context is the negotiable type of rights when developing/acquiring reusable software components.

The level of software rights the government negotiates under a contract should be determined based upon what rights are needed in the future for reuse to work and also so the responsibility of maintenance is taken into consideration. Some experts argue the government should always have unlimited rights to software it paid to have developed. Others argue that to facilitate software reuse, the government should always obtain "restricted rights". If the government were to be more creative when negotiating rights, then the government's and industry's interests could be protected through techniques such as: escrow of rights, negotiated periods of contractor rights exclusivity, incentives and other means [CARDSa]. Detailed guidance in determining when to negotiate what types of rights, depending upon the software reuse strategy, is discussed in detail in the CARDS *Acquisition Handbook* [CARDSa].

Liability can be based on contracts, tort, and statutes. Liability in contracts can result when one party breaches a term of the contract (includes warranties) [REUSE93b]. Based on the common law of tort, one has a duty to provide a certain "standard of care" to another when it is expected of them (includes strict liability, e.g., unreasonably dangerous product). Of course, if laws (statutes) are not obeyed, one can be at risk to pay penalties as covered under law. Thus, the main concern regarding risk/liability in the component development context is reducing liability by making sure laws are followed and taking care in performing duties, as well as by using license and usage agreements. In addition, product liability can be reduced by incorporating a product liability strategy, as well as relying on legal protections in agreements and contracts [ARMOUR93].

Once the particular needed or wanted rights are chosen, they need to be protected when reused or distributed, especially if distributed via a reuse library to possibly many subscribers. Software and its related documentation, such as designs and specifications, should be protected by using both agreements and markings on the software [FEDPUBS][CARDSh].

Agreements can be written to cover all users, such as commercial shrink-wrap licenses, or terms and conditions of agreements can be negotiated on a case-by-case basis. If software and related documentation will be distributed via a reuse library, it makes more sense for the developer to negotiate one agreement with each library, rather than negotiate an agreement with every user who accesses each of the libraries. Some terms and conditions of an agreement will be specific to the end users and other terms and conditions will be specific to the library. Thus, any agreement with a library must have terms in it pertaining to both.

Some terms and conditions applying to a library include, but are not limited to the library's policies and procedures for:

- submitting components, including formats,
- evaluating components and distributing results of evaluations,
- collecting and forwarding fees; requesting comments from end users,
- specifying who will maintain the component and the procedures for maintenance.

Library agreements are discussed in detail in the CARDS *Model Contracts/Agreements* document [CARDSh].

Some terms and conditions applying to end users include, but are not limited to (Refer to Appendix C for a description of these terms):

- using, duplicating, disclosing, modifying the source code (e.g., derivative work),
- warranties (or lack of),
- indemnification,
- hold harmless,
- disclaimer,
- nondisclosure.

3 SOFTWARE COMPONENT DEVELOPMENT

3.1 Introduction

The commercial market for reusable software components started with Fortran math and statistics function libraries. In the 1980's reusable components were based on commercially available abstract data types and Ada, such as the Booch components and the GRACE components. Late in the 1980's the object-oriented (OO) movement produced C++ class libraries that were sold by separate vendors as products, and software ICs/class libraries that were sold by the OO language compiler vendors (e.g., Objective-C and Eiffel). Most of these were generic, small scale data structures and graphics components meant for horizontal domains. Interest in developing vertical domain components has recently emerged, but there are still obstacles such as the lack of standards and few organizations have domain-specific and software technology expertise [BUCK93].

A goal of this handbook is to stimulate the development and commercialization of large-scale components for vertical domains. Such components are similar to stand-alone commercial-off-the-shelf (COTS) application programs marketed as separate products (some of these COTS products are suitable as reusable components with little modification). Government-off-the-shelf (GOTS) and public domain software can and should fill some of the needs for reusable components. The main differences between currently available components and the ideal catalog of reusable components are:

- Reusable components need to be "open" (i.e., be designed to accept and transfer data and control messages in a standard way so they can be easily integrated into large systems).

- Reusable components for engineering and military domains need to be developed to supplement the existing pool of common marketplace office automation components.

- The source code for COTS components is often not available and is not written in Ada as is often required on government contracts.

Reusable components include more than just source code. Requirements, design and test components should be developed along with code components. They can be packaged with the code component or be supplied as an independent entity. In the past, requirements, design, and test information were sometimes provided with the code as text documentation. The goal is to provide this information in a standard and machine processable form that is easier to reuse.

For a software component to be reused it must satisfy a perceived need by the user and must satisfy a perceived level of quality. At a minimum, the user must understand the function of the software and must assume the use of the software will not cause unpredictable effects within the software system where it will be used.

A method widely used throughout the DoD to achieve high standards of predictability and quality is the guidelines and practices described in the Capability Maturity Model for Software

(CMM) developed by the Software Engineering Institute (SEI) [PAULK93a][PAULK93b]. This methodology presupposes that the quality and predictability of the software is the result of the processes used to create it and focuses upon management/development process definition and control. The best advice is to use sound engineering development practices following an accepted standard for software development management/documentation and describe, in the software documentation, the standards and processes followed.

This chapter of the handbook discusses the process of developing reusable components. During domain analysis and modeling (Section 3.2) the groundwork is laid for developing the generic architecture (Section 3.3) and defining component requirements and specifications (Section 3.4). The components are then developed (Section 3.5) and tested (Section 3.6). After test completion, the components are made operational and maintained (See Section 3.7). An underlying theme is that the capabilities of Computer Aided Software Engineering (CASE) and reuse tools should be used to support the development of components. Figure 3-1 illustrates the software component development process described in this chapter.

3.2 Domain Analysis and Modeling

The practice of considering a domain and the products to be developed for a domain is known as domain engineering. "Domain engineering refers to the techniques (i.e., methodologies) used to analyze and model an application domain, and construct reusable components based upon these analyses and models." [CARDSd] Domain analysis is an important early step in domain engineering and refers to any activity helping to collect and organize information about the subject, or domain, being studied. For the purposes of component developers, this generally comes down to studying existing components in their area of interest, technical and business issues related to such components, and related available requirements (for future components or systems).

To maximize the benefits of domain analysis, the knowledge obtained is formally expressed via a model. At the very least a model provides a means of knowledge communication; ideally it is also the springboard for the design and even generation of future components. In the case of CARDS, a version of the domain model is the organizing principle in the library and the basis of the ability to compose separate components into meaningful subsystems.

This section describes domain analysis efforts of component creation:

- domain planning (Section 3.2.1),
- domain analysis, including sample methods (Section 3.2.2),
- domain engineer tasks and responsibilities (Section 3.2.3),
- component creator tasks and responsibilities (Section 3.2.4),
- SPO engineer tasks and responsibilities (Section 3.2.5),

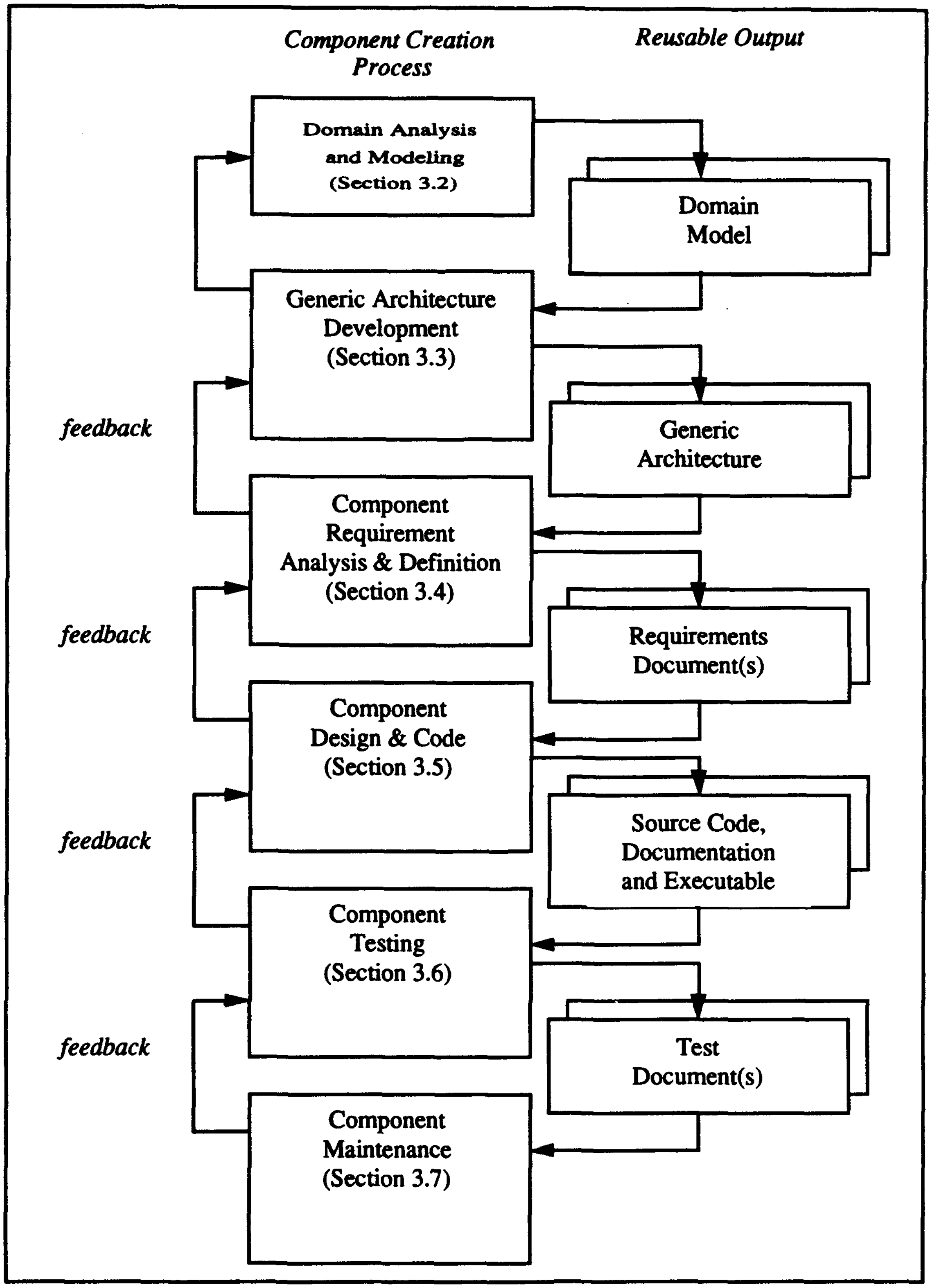
Component Creation Process
Reusable Output
Domain Analysis and Modeling (Section 3.2)
Domain Model
Generic Architecture Development (Section 3.3)
Generic Architecture
Component Requirement Analysis & Definition (Section 3.4)
Requirements Document(s)
Component Design & Code (Section 3.5)
Source Code, Documentation and Executable
Component Testing (Section 3.6)
Test Document(s)
Component Maintenance (Section 3.7)
feedback
feedback
feedback
feedback

Figure 3-1 Software Component Development Process

- Evaluation techniques for domain analysis (Section 3.2.6).

3.2.1 Domain Planning

DoD is in the process of identifying and setting up domains. The *DoD Software Reuse Vision and Strategy* [DoD92] envisions the services establishing centralized reuse organizations. There are reuse initiatives within the services as well as central organizations, such as DISA. The *DoD Domain Definition Report* [CSRO92] provides a high-level representation of existing DoD domains. Figure 3-2 is taken from that document. Domain engineers should consider areas of commonalities between domains in different services and other potential government domains including:

- Department of Energy (DOE),
- Department of Transportation (DOT),
- Federal Aviation Administration (FAA),
- Central Intelligence Agency (CIA),
- National Security Agency (NSA),
- Federal Bureau of Investigation (FBI),
- Department of Health and Human Services,
- Department of Treasury/Internal Revenue Service (IRS),
- Postal Service.

Besides these government domains, there are many commercial domains covering similar applications as well as very different applications like engineering, science, medical, and business. The point is that reusable components could be applicable to many different government and industry domains if there are commonalities in the functionality required. This is especially true in the growing world of "open systems".

The rest of this section discusses the following important domain planning issues:

- domain management (Section 3.2.1.1),
- legal issues (Section 3.2.1.2),
- acquisition (Section 3.2.1.3),
- status of the technology (Section 3.2.1.4),

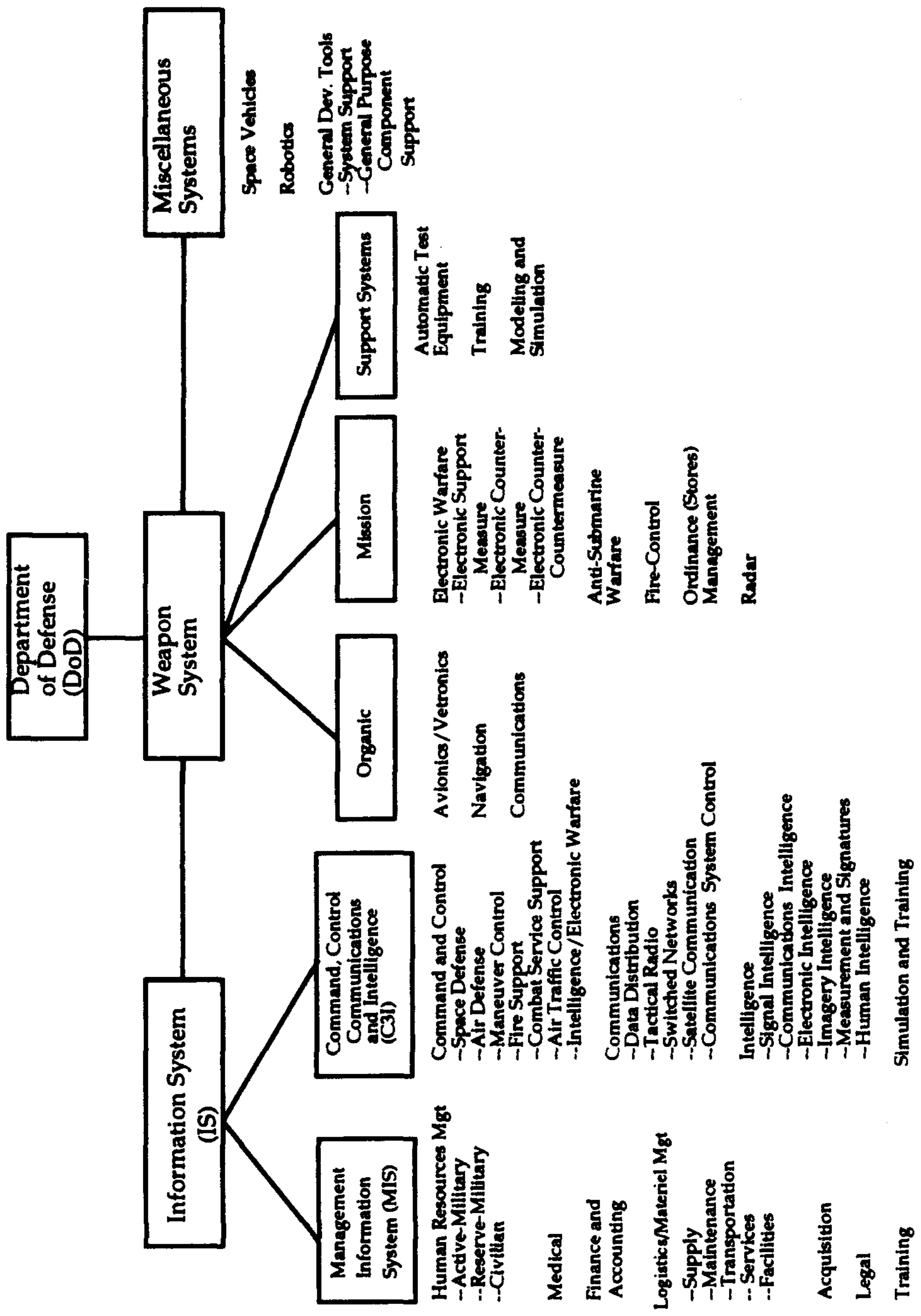

Figure 3-2 DoD Domains

- customer forecast (Section 3.2.1.5).

3.2.1.1 Domain Management

An important domain planning factor is to identify domain management issues/concepts, e.g.:

- How are domains managed? Examples include:
  - No central management (project by project).
  - Central domain management but not planned.
  - Centrally managed and reuse management plans based on product line.
- Have the domains for the appropriate organization been identified?
- Are domain engineering efforts identified and prioritized in product line plans?

The Army is managing domains through Program Executive Offices (PEOs). The Air Force has designated Reuse Project Officers (RPOs) to help organize and manage domain specific reuse.

3.2.1.2 Legal Issues

Legal issues affect the development of domain components. The primary legal issues pertain to the ownership rights of the components and who assumes responsibility (i.e., liability) for these components. What ownership rights will your customers want? What ownership rights will reusers (on any derivative work) want? What rights will original developers have for your components which was reused using the original developer's work? The current data rights laws and government policies should be examined to determine how they will effect your organization's acquisition, technical, and reuse goals. For information on legal issues and contractual guidelines, see the CARDS *Direction Level* [CARDSb], *Acquisition Handbooks* [CARDSa], and Section 2.7 of this handbook.

3.2.1.3 Acquisition

The CARDS *Direction Level* [CARDSb] and *Acquisition Handbooks* [CARDSa] outline existing disincentives/inhibitors and possible incentives the government could provide. When planning the strategy of a particular domain, incentives and disincentives must be considered. Ideally there will be a large pool of existing COTS and GOTS components. If not, incentives for the development of COTS components may need to be planned. Deals and incentives may need to be pursued to obtain GOTS components as well.

3.2.1.4 Technology

The state of domain's technology is a major planning issue. Analyze current and future software development practices. Are current systems in the domain:

- Developed with obsolete technology?
- Exhibiting poor reliability?

Forecast product evolution as a function of technology and market needs [SPC92b]:

- What kinds of functionality do existing systems in the domain have?
- What kinds of technology advances will effect this domain?
- What kinds of military threat are projected for this domain?

Identify technology areas the organization considers key to its future competitiveness and in which it invests research and development (R&D) funds.

Ascertain the domain's maturity. Is the domain still in the stage where research is generating new solutions rapidly? If so, this domain may not be mature enough to benefit from reuse.

Another technological market indicator is the level of standardization within a particular domain [SPC92b]. Reuse potential can improve with increasing levels of standardization due to the formality, increased stability and increased understanding of a domain. Moreover, if a domain is more stable, it is easier to predict future developer and end user needs. Besides looking at standardization to determine stability, the rate of change in technology within the application domain can also indicate stability. Rapidly changing technology can create an unstable and thus an unpredictable environment. Technology dependencies in the domain should also be determined. In addition, the age of a domain and existence of some sort of domain model and generic architecture, the rate the domain is maturing, and the "product life cycle" phase the domain is in, are other aspects to consider [BOEING92].

3.2.1.5 Customer Forecast

Planning a domain is like doing a market study. The customers are application engineers and actual users of systems. As in any other market study, the possible opportunities and constraints must be weighed against the organization's capabilities to determine the risks associated. The possible domain risks must be identified, assessed and determined if they can be minimized. When examining risks, each of the above indicators (e.g., domain status, legal issues, acquisition, and technology) is considered.

Forecasts can be estimated utilizing the conventional marketing techniques. The demand should be estimated for each applicable type of component (i.e., domain model, architectures, designs and implementation components). These forecasts should take into consideration the demand for products in terms of new or existing core capabilities/products that should be developed with respect to the component types [SPC92b].

There are some additional points, specific to software reuse, to be considered. Customers may demand different versions of similar products. A customer may demand/want software that can be upgraded, managed and applied to new problems [SPC92b]. The bottom-line, of course, is to completely understand the plans and constraints of customers' present and future needs. In estimating economic factors (e.g., return on investment and investment costs) there are additional factors introduced by software reuse. First, initial investment costs could include: domain analysis, reuse-related training, and new software/hardware. Next, cost estimates of components may differ from past experiences, since the existing software costing models do not take software reuse into consideration [SPC92b]. Lastly, in estimating break even cost, the break even number of components and systems should also be calculated.

3.2.2 Domain Analysis

There are no hard and fast rules about how domain analysis should be done. The important points are to carefully bound the domain being considered, to consider the ways the systems in the domain are alike (suggests required characteristics) and the ways they differ (suggests optional characteristics), to organize an understanding of the relationships between the various elements in the domain, and to represent this understanding in a useful way.

> *"A more dramatic improvement of the reuse process results when we succeed, through domain analysis in deriving common architectures, generic models or specialized languages that substantially leverage the software development process in a specific problem area. How do we find these architectures or languages? It is by identifying features common to a domain of applications, selecting and abstracting the objects and operations that characterize those features, and creating procedures that automate those operations. This intelligence-intensive activity results, typically, after several of the "same kind" systems have been constructed. It is then decided to isolate, encapsulate, and standardize certain recurring operations. This is the very process of domain analysis: identifying and structuring information for reusability." [PRIETO-DIAZ90]*

The input to domain analysis includes knowledge from domain experts, documentation and code from one or more existing systems, and handbooks and reports describing general domain principles and system design. Early identification and inclusion of domain experts in domain analysis greatly reduce the risks involved in bounding the domain and in defining the vocabulary and concepts of the domain. Common products produced by most domain analysis methodologies include [HOLIBAUGH92]:

- entities, concepts, categories,
- data (variables and constants),
- functions, services, features,
- relationships (e.g., specialization and aggregation),
- constraints,
- vocabulary (consistent terminology).

Domain analysis approaches have the goal not only of increasing the understanding of the domain, but of making that understanding communicable, and sometimes machine manipulable. To do this the information obtained and organized during the analysis must be put into some sort of formal model. There are many types of models. Some approaches to domain analysis produce several different models, each intended to highlight a different aspect of domain information or to be used in a different way. Models commonly use representations such as [HOLIBAUGH92]:

- object-oriented,
- entity-relationship-attribute,
- artificial intelligence (e.g., semantic networks, frames and rules).

There are many domain analysis approaches currently being developed and in practice. This section briefly describes the following domain analysis approaches:

- CARDS/PRISM Domain Analysis (Section 3.2.2.1),
- Organization Domain Modeling (Section 3.2.2.2),
- Feature Oriented Domain Analysis (Section 3.2.2.3),
- STARS Domain Analysis (Section 3.2.2.4),
- DSSA Domain Analysis (Section 3.2.2.5),
- Bailin/KAPTUR Domain Analysis (Section 3.2.2.6).

3.2.2.1 CARDS/PRISM Domain Analysis

CARDS is using the results of PRISM's domain analysis and the initial library components identified and/or built by PRISM. The CARDS taxonomy and high-level architecture are based on PRISM's work , which in turn is based on a combination of general experience in the field of command centers and specific experience building prototypes for reuse. CARDS/PRISM did not initially follow a well-defined domain analysis process, so the following description focuses more on the created model rather than the process used to create it.

The CARDS command center model is being worked from both top down and bottom up perspectives. From the top there are two lines of development. First, information about the military functionality required of a command center from the DISA *Command Center Design Handbook* (CCDH) [CCDH87] is encoded in the model. These command center requirements have been allocated to specific component types or subsystems included in the model. Eventually CARDS will support system composition (see Appendix E) based on the user choosing requirements rather than subsystems. Second, CARDS has adopted a PRISM software architecture. This architecture may be visualized as an aggregation tree, with very high-level subsystems at the top, each of which is broken down into lower level subsystems, and so forth until you reach the level of individual components, wrappers (software written to include COTS or GOTS tools in an integrated system), and interfaces. The higher levels of this work come from the CCDH. Lower levels were developed by PRISM. Integrated into this breakdown of systems is a Technical Reference Model (TRM). PRISM and CARDS adopted the reference model presented on the DISA TRM [DISA92b]. This reference model has two aspects: it is a collection of standards to be used in system development and it provides a way to organize types of software.

The lower level details of the CARDS model are based on PRISM's experience in assembling combinations of PRISM components to work together to gain the most flexible configurations of prototype components. The CARDS command center model includes a very concrete representation of a specific set of components and the ways in which they may be harnessed together to accomplish large tasks. This supports CARDS emphasis on providing practical tools to actually compose systems to be reused. However, in the long run more top down analysis is needed to insure that this low level, practical information is embedded in a structure effectively supporting the widest range possible of command center structures. CARDS seeks input from additional sources to assure the generality of the software architecture represented in the library model. Recently CARDS has begun using the Organization Domain Modeling (ODM) approach.

3.2.2.2 Organization Domain Modeling (ODM)

ODM is a general method for domain analysis and modeling, including a structured set of work products, a tailorable process model and a set of modeling techniques and guidelines [STARS93]. ODM processes and work products have been structured based on the reuse process model of the *Conceptual Framework for Reuse Processes (CFRP)* [STARS92]. ODM is a collaborative, team-based modeling effort involving all the stakeholders in the design. ODM supports the ability to map points of commonality and difference without trying to "work" or resolve alternatives too rapidly. Multiple views of the same information are supported by the ODM methodology. ODM supports both "descriptive" and "prescriptive" modeling. Descriptive modeling, in simple terms, examines existing systems and known requirements and finds commonalities and differences between members of an "exemplar set." Prescriptive modeling represents decisions and commitments to functionality to be supported and expresses the range of variability of these decisions.

3.2.2.3 Feature Oriented Domain Analysis

Feature Oriented Domain Analysis (FODA) is a domain analysis methodology developed by the SEI [KANG90][COHEN92]. The primary focus of the method is the identification of prominent or distinctive "features" of domain software systems. These features are user-visible aspects or characteristics of the domain. The FODA process is divided into three phases [PETERSON93]:

1. Context analysis where domain analysts interact with users and domain experts to scope the domain. Context analysis produces a context model.

2. Domain modeling to produce a features model, an entity-relationship model, a data flow model, and a finite state machine model.

3. Architectural modeling which maps architectural concepts onto the domain model to produce a generic design.

3.2.2.4 STARS Domain Analysis

A STARS domain analysis approach has been formulated [PRIETO-DIAZ87]:

1. The preliminary phase consists of defining and scoping the domain, identifying sources of information and defining the specific approach to domain analysis to be used for the domain in question.

2. The domain analysis proper consists of:

   A. Identifying relevant objects and operations in the domain.

   B. Finding a general description of the classes of objects in the domain, usually expressed in frames.

   C. Classifying is constructing a taxonomy of the domain objects, expressed as relationships between the frames.

3. Post domain analysis includes the creation of new objects (components) for the domain and the establishment of reusability guidelines.

Prieto-Diaz's approach to domain analysis involves the development of a faceted classification scheme bolstered by term thesauri and a conceptual distance graph. Facets are characteristics by which the domain objects may be described. Associated with each facet is a list of terms defining the possible values for that facet. The term thesauri means certain terms are synonyms and these synonyms can be used in searches. The conceptual distance graph is a mechanism for specifying how similar any two classified objects are. This provides a way to give a "next best result" to a search that does not result in an exact match. This domain analysis approach incorporates both bottom-up and top-down aspects. The process by which the faceted classification scheme is derived is bottom-up. The specification of the conceptual distance graphs is one type of top-down information.

3.2.2.5 DSSA Domain Analysis

ARPA's Domain Specific Software Architecture (DSSA) Program is concerned with the overall environment in which software development takes place and with leveraging the concepts of domain engineering for long-term productivity improvements [METTALA92]. Several industry and university teams are working on a variety of approaches in different domains. In general, the DSSA approach is divided into three phases:

1. Domain analysis where participants construct various types of domain models and a component-based architecture.

2. Define DSSAs via architecture definition languages, including module interconnection formalisms. Prototype systems are developed by composing a set of components to satisfy a specific system architecture based on the domain architecture.

3. Building systems in military environments.

DSSA recognizes three system levels: a domain development environment, a domain-specific application development environment and an application execution environment. A domain architect develops the domain development environment. From this environment an application engineer takes a reference architecture, components and development tools to create the domain-specific application development environment. The product coming out of the application development environment is an architecture instantiation an operator runs in the application execution environment.

The domain architecture constrains, and thus guides, development of components. These constraints simplify the component requirements (they don't have to be all things to all people) and thus ease development and later evolution. To support systems evolution, there must be feedback from the application execution environment to the domain development environment.

3.2.2.6 Bailin/KAPTUR Domain Analysis

Sidney Bailin, of CTA Incorporated, is one of the key proponents of a domain analysis process geared toward the use of a tool developed by CTA for NASA, KAPTUR (Knowledge Acquisition for Preservation of Tradeoffs and Underlying Rationales). The process and the tool support a reuse-based software development environment [BRACKEN]. While FODA uses a feature-oriented approach, the KAPTUR-based approach introduces the concept of a "distinctive feature", which is described as any distinctive aspect (whether or not visible to the system user) of a system or reusable software objects differing from common or recommended practice, or represents a significant development decision [BAILIN92]. The uniqueness of the KAPTUR-based process is that it captures and represents knowledge about these features, knowledge concerning design decisions, tradeoffs, and rationales, that other domain analysis processes do not do effectively.

The context for the KAPTUR-based domain analysis process is founded within an improved process for software development consisting of reuse-based engineering. That process includes domain engineering and application engineering. Domain engineering consists of domain analysis and domain implementation. The KAPTUR-based process supports the domain analysis portion of this environment. The purpose of the domain analysis is to identify and capture knowledge about domain commonalities and variations (parameters of variations). The domain analysis must result in a domain knowledge base that is organized, maintainable, and presented in such a way as to prevent user information overload [BAILIN93].

KAPTUR is the tool for capturing, organizing, maintaining, and representing the domain knowledge base. The process includes concepts of object-oriented modeling, feature modeling, and case-based reasoning. Object-oriented modeling recognizes the link between analysis and object-oriented design and the advantages object-oriented methods give in software development. The feature modeling aspect distinguishes between engineering choices/alternatives and the rationale [illegible]ose choices. Features are those system attributes sufficient to characterize the system/domain aspects and their variations for each domain stakeholder (e.g., user, designer, and acquirer) in the domain. Constraints, an important aspect of architectures, can be represented in the features in terms of their being mandatory, optional, variant, or exclusive. Features can be a decision recording mechanism in descriptive modeling or a requirements specification mechanism in prescriptive modeling. Classifications of features suggested by [BAILIN93] include:

- operational,
- interface,
- performance,
- functional,
- development methodology,
- design,
- implementation.

CASE-based reasoning places emphasis on descriptive modeling, or the creation of a database of legacy or exemplar systems.

The KAPTUR-based domain analysis process involves both descriptive modeling steps, where the analyst looks backwards at existing or legacy systems, and prescriptive modeling, where the analyst looks forward at new implementation alternatives. The descriptive modeling steps involve identifying and scoping the domain and analyzing domain information. The modeling approach (view, diagram) is determined in this second step, and the view(s) selected depends on the domain influences. The modeling approach/view is not determined beforehand. The next step in the KAPTUR-based process is a transition from descriptive to prescriptive modeling and involves the creation and maintenance of a domain model. The prescriptive modeling steps validate the domain model and identify automation opportunities. The action to identify opportunities is to look for recurring patterns, as a technique to analyze opportunities for reuse.

Object-oriented modeling is an important aspect of the KAPTUR-based process and KAPTUR provides the formalism representing the domain knowledge base. This is done through the use of modeling diagrams. In KAPTUR Release 1.0, these include the Entity Relationship Diagram, Classification Diagram, Assembly Diagram, Dataflow Diagram, Object Communication Diagram, State Transition Diagram, and the Stimulus Response Diagram. Each diagram or architectural view answers a specific question about the domain being modeled (e.g., for the Entity Relationship Diagram, what are the user roles, objects, and data items participating in the system, and how do they interrelate? [BAILIN93]). If a question relating to a particular diagram or view is not relevant, then the diagram is probably not very useful to the domain knowledge base.

Elements in the KAPTUR tool have been set up to capture and organize the knowledge gained from the domain analysis process. This is the strength and the uniqueness of the KAPTUR-based process. The elements organized include [BRACKEN]:

- architecture views (modeling diagrams) and free text descriptions of the particular architectural view,
- objects and related annotations or descriptions existing in the architectural views,
- features about the architectural view, and for each feature, there is information organized about the decision represented by the feature, the tradeoffs needing to be considered in making the decision about the feature, and the rationale associated with decisions made relative to the feature. In addition, the tool identifies alternative domain architectures not having the feature in question.

3.2.3 Domain Engineer Tasks and Responsibilities

The domain engineer's domain analysis tasks begin at the start of the component creation process. Since domain analysis can be thought of as the definition of the "problem space", this work must be accomplished before the solution can be found. The tasks and responsibilities of the domain engineer during this component creation phase include all domain analysis defined tasks.

3.2.4 Component Creator Tasks and Responsibilities

The component creator role depends on the level of domain knowledge the component creator has to contribute to domain analysis and the organizational/contractual relationship the component creator has with the domain engineers. The component creator needs to be kept informed of all progress and perhaps be included whenever possible in the review of intermediate documents created in this phase.

From a purely profit-oriented business point of view, a company may develop its own analysis of a government or commercial domain to determine component development return-on-investment.

3.2.5 SPO Engineer Tasks and Responsibilities

SPO engineer roles depend on the level of domain knowledge SPO engineers have to contribute to domain analysis and how much the domain engineer decides to delegate to SPO engineers. The main responsibility of SPO engineers is to assist in the development of the acquisition documentation. [CARDSc] describes the responsibilities of SPO engineers during the acquisition phase of a contract (which includes domain analysis). As the domain analysis progresses, SPO engineers continually monitor the feasibility of component reuse (where component includes the reuse of all aspects of system design, requirements, source code and testing).

3.2.6 Evaluation

To determine if domain analysis and modeling are complete for the next phase of component creation (See Section 3.3), both the domain and SPO engineers must evaluate the following goals:

Goal: To determine if there is a sound basis for defining component requirements, confirm that a documented systematic process for domain analysis and modeling is being or was applied.

Questions:

- Is the process documented in a plan or some other form?
- Is the process supported by automated tools?
- Does the process include peer reviews of domain analysis products?
- Does the model provide for feedback from other phases of component development?

Goal: Confirm that an adequate model has been developed. An adequate model provides a framework for component development.

Questions:

- Can the information represented in the model be used in the:
  - Requirement analysis and definition process?
  - Component development process?
  - Component qualification process?
  - System development process?
- Does the model support communication among component developers and users?
- Has a consistent domain vocabulary been defined and documented?
- Does the model completely cover the scoped domain?
- Has the model been independently validated?

- Does the model representation support automation of the:
  - Component development process?
  - System development process?
- Does the model representation support the specification, design and code of reusable components with reusable components?

3.3 Generic Architecture Development

There is a great deal of interest and research in the area of software requirement/design reuse and generic architectures. The area is rapidly evolving. It is widely recognized that a generic architecture is a major facilitator for software reuse.

3.3.1 Generic Requirements

Traditional software reuse has concentrated on reusing code and design segments within a system or across systems. It is widely recognized that reuse of requirements offers an even larger potential payoff. If one views a system or subsystem as a black box meeting certain requirements and if these systems are built by choosing from the requirements that have corresponding implementation components, we can potentially reduce the cost of developing a system/subsystem to near zero. This concept of "megaprogramming" is being strongly espoused by DoD and is already being enforced in the acquisition of some systems. Generic software system level requirements are sometimes considered to be part of the generic architecture (the architecture should at least trace to a set of requirements). These requirements should be derived primarily from the domain model. Possible representations for the requirements include:

- natural language (e.g., plain English),
- formal language with a specified syntax and semantics (e.g., predicate logic or application specific languages),
- model (e.g., semantic network).

Natural language is the easiest for the end user of a system to understand. However, much has been written about the perils caused by the ambiguities of natural language for software specifications. To promote reuse of requirements there must be a move toward more formality in requirements. To automate reuse these requirements need to be machine processable and be linked to their implementations via a model.

Machine processable requirements include objects, actions, logic, and timing/sequencing. In limited domains these elements can be defined and strung together with a specified syntax (like a programming language). The resulting formal language can be used to generate code, and to document and test components with compiler and knowledge-based system technology. Fourth generation languages are an example of this approach and have shown to be useful in restricted tasks that are commonly applied (e.g., spreadsheets). Application specific languages have been applied with much success to very specific customized application domains. The formal language could also be implemented with templates and menus rather than statements.

3.3.2 Generic Architectures

The key to design reuse is to develop and document an architecture that is generic enough for most systems in the domain and that supports the reuse of code components in the domain. The architecture should:

- anticipate variations in domain systems,
- have a coherent style that reduces maintenance efforts,
- use familiar patterns to facilitate understanding,
- include rationale/trade-off documentation.

There are a variety of definitions for software architecture:

- organizational structure of a system or component [IEEE STD 90],
- components and connectors [GARLAN93a],
- elements, form, and rationale [PERRY92],
- a specification for the assemblage of components [BRAUN92].

Although there are disagreements as to exactly what is included in a software architecture, many of the definitions point to an architecture which is a high-level design that was developed/applied in a disciplined way. This architecture defines the relationships among the major components of the system with system specific details abstracted away. Besides the architecture, there is also detailed- or low-level design focusing on component algorithms.

Academic researchers are currently studying and classifying software architectures (similar to the way a biologist would study species of plants or animals) [GARLAN93a]. Hopefully this leads to the identification of common styles (idioms) and system patterns. The long term goal is to develop guidelines for applying these styles and patterns in new/reengineered systems to promote reuse. Some of the main styles and patterns identified so far are:

- data flow style:
  - batch sequential - each step runs to completion,

- pipes and filters - linked stream transformers.

- call and return style:
    - main program and subroutines - traditional functional decomposition,
    - hierarchical layers - well defined interfaces and information hiding (e.g., kernels and shells),
    - Object-oriented systems - abstract data types with inheritance.
- independent components style:
    - communicating processes - asynchronous message passing,
    - event systems - implicit invocation.
- virtual machines style:
    - interpreters - input driven state machine,
    - rule-based systems - rule based interpreter.
- data-centered systems:
    - transactional database systems - central data repository/query driven,
    - blackboards - central shared representation/opportunistic execution.

Currently it appears that an architecture having a mixture of object-oriented [BOOCH] and event system [GARLAN93b] characteristics is best suited for supporting reuse of design and code in most domains. Object-Oriented concepts can be extended to large architecture level components. Event systems are an outgrowth of CASE tool integration and other research efforts. Event systems allow for looser coupling between components by using an event manager to implicitly invoke operations on objects and thus eliminate name dependencies. PRISM is currently using an architecture based on object-oriented and event system mechanisms.

The Object Management Group (OMG) is working on the Common Object Request Broker Architecture (CORBA) which will help standardize object-oriented event system architectures [OMG92]. CORBA has an object model attempting to standardize object management services across heterogeneous platforms and establish common facilities (standard general utility objects - e.g., editors, help facilities, and email). This is done by establishing standard object interfaces (signatures) including operations and parameters. The object model would promote extensive reuse of general objects. CORBA also has a standard model for an object request broker (event manager). CORBA should prove to be a powerful reuse approach.

A domain-specific software architecture (DSSA) is a software architecture based on the actual and projected commonalities and differences of domain systems. DSSAs are sometimes called generic architectures (e.g., PRISM). A DSSA is a reusable design component providing the basis for other reusable design and code components. DSSA development should be started during domain analysis (some consider a DSSA to be part of the domain model). The development of DSSAs is an active research topic. ARPA's DSSA Program is developing the concept of a DSSA in various military domains [METTALA92]. DSSA is focusing on domain analysis, architecture description languages, tools, and processes. The SEI has built a DSSA from a domain model and a special object-oriented architecture style called Object Connection Architecture (OCA) [PETERSON93]. OCA defines a set of domain independent object types and uses templates to standardize implementation. Other researchers are developing DSSAs called object-oriented frameworks [BUSCHMANN93][NIERSTRASZ92]. An object-oriented framework is both a reusable architecture and an architecture supporting reuse of components. The framework defines a basic set of component classes and their interconnection structure. Also, basic operations and data (signatures) are defined at an abstract level for the domain. An object-oriented framework can be designed to be adaptable and flexible so new objects or subsystems can be grafted in or removed.

DSSAs are documented in multiple views or models. The SEI advocates model-based software engineering (MBSE) [WITHEY93] where the reuse of models is central to the development of systems in a domain. Various forms of design models that can be reused include:

- data flow diagrams,
- flow charts/control flow diagram,
- structure charts,
- object-oriented diagrams,
- state diagrams,
- process timing models,
- memory allocations,
- algorithms/program design language,
- constraints,
- text descriptions of architecture parts and reusable components,
- text discussions of rationales and tradeoffs,

- interface specifications (syntactic and semantic) including Ada bindings to programming languages and specific software in another language,
- formal or informal interface standards like:
    - POSIX (Unix OS),
    - X11 (window system),
    - Motif, OpenLook (window managers),
    - GOSIP (network services),
    - SQL (DBMS),
    - SGML (text encoding),
    - GKS (graphics),
    - Postscript (printer).

Graphic design models can be developed and stored with CASE tools (see Chapter 4). The CASE tool representations and methodologies should be suitable for the domain. For example, in some domains, complex sequences of states and actions are common, so the tool should facilitate the representation of these applications. Template-like diagrams for a domain can be created with some CASE tools. These generic domain diagrams represent design patterns which can be specialized for a new system in the domain; thus reusing previous design efforts and promoting standardization in the domain. Another approach is to store CASE tool representations of complete existing system designs. These complete designs can then be modified for a new system in the domain. Retrieving these types of design components is difficult because it is hard to describe/categorize these design cases. KAPTUR is a tool facilitating the storage and retrieval of system specific design components for future modification. Some CASE tools have facilities for simulation and evaluation which are useful for facilitating system understanding and reuse.

Interface definitions may be approved standards (e.g., ANSI, ISO, NIST, IEEE, and FIPS), default industry standards (e.g., operating systems and network services), or domain-specific standards (e.g., message formats). Most interface standards define a set of services provided by a component and a syntax for accessing these services. This syntax ranges from language independent to language specific. Interface standards are especially important for a reuse based development process. They should be defined for a project even if the basis architecture did not originally establish them. The set of architecture standards is sometimes called a reference model. Open systems facilitate the reuse of components across hardware platforms. POSIX, a standard Unix interface, is a key step toward open systems. GOSIP (Government Open Systems Interconnection Protocol) is an emerging standard for network services and another important part of open systems. Data related standards (e.g., SQL and SGML) facilitate interfaces between reusable components. User interface standards (e.g., X11 and Motif) promote reuse of significant amounts of code necessary for humans to effectively interact with the system. The DISA *Technical Reference Model* [DISA92b] has more information on current and emerging standards.

Reengineering is an excellent source for reusable design models and components, especially when there are well designed systems existing in the domain. Current tools provide some automated support for extracting high- and low-level designs directly from a system's source code, because design documentation often does not exist or is not consistent with the code. Then the design fragments can be restructured with automated and manual methods into reusable design components for a domain. Several systems in a domain may be used as a source, especially when developing DSSAs. Extracting commonalities in the designs for a domain will result in design components that are more reusable. Conversely, reengineering of systems should be done in the context of a domain model and a DSSA [FEILER93].

3.3.3 Domain Engineer Tasks and Responsibilities

The domain engineer is responsible for the development and evolution of the generic architecture. Since this is a large and difficult task, the domain engineer will often form a team of systems and software engineering experts to help. The domain engineer may also draw on the knowledge of the SPO engineers and component creators.

3.3.4 Component Creator Tasks and Responsibilities

The role of the component creator depends on the level of domain knowledge and software architecture expertise the component creator has to contribute and the organizational/contractual relationship the component creator has with the domain engineers. The component creator should be kept informed of all progress and be included in the review of intermediate documents created in this phase.

3.3.5 SPO Engineer Tasks and Responsibilities

The role of the SPO engineer depends on the level of systems and software architecture knowledge the SPO engineer has to contribute and how much the domain engineer decides to delegate to the SPO engineer. It is import that the SPO engineer be involved, since the generic architecture will greatly affect the systems being built or reengineered in the future.

3.3.6 Evaluation

As part of the generic architecture development phase the following goals should be reviewed by the SPO engineer and the component creator. These goals are specific for software reuse and are new to many organizations; therefore, it is advantageous to review these goals periodically.

Goal: Ensure the generic architecture is reusable across domain systems.

Questions:

- Does the generic architecture allow for variations in system architecture and detailed design?
- Has the generic architecture been modeled adequately? Does it support the development of components?
- Does the architecture account for current and emerging standards?
- Does the generic architecture have adequate tools to support application engineering?
- Does the generic architecture have documentation on rationales and tradeoffs?

3.4 Component Requirements Analysis and Specification

At this point a domain model and generic architecture should have been developed and now it is time to define actual candidate components. In parallel with investigating domains, current and planned reuse libraries should be surveyed to determine what components already exist and where the largest gaps lie.

3.4.1 Domains as Sources of Component Requirements

The generic architecture should contain at least an initial bundling or partitioning of system level requirements or features as well as intercomponent interfaces. The mapping of a domain model and generic architecture to actual implementation components is still an area of active research. Some existing design methods and architecture style definitions (e.g., OCA) provide some heuristics. The choices may be constrained by the cost effective availability of COTS, GOTS, and reengineered components (see the next section) to fill in part of the architecture. However, even with a detailed domain model and generic architecture there are still detailed requirement decisions to be made about each component. Each component developed for the domain should have a specification (see previous section for a discussion on representing requirements). To maximize the usefulness of the candidate components, it is wise to consider related domain requirements. With adequate planning, a software component for one domain may be suitable for use in other domain applications or libraries with little or no modification.

3.4.2 Reuse Libraries as Sources for Component Requirements

Analyzing existing and planned reuse libraries is another important step for determining the requirements of reusable components that are candidates for development. Of the existing reuse libraries, what domains are represented or not represented? Potential areas for consideration could be those needed, but for which libraries and components do not currently exist. What domains do new or planned libraries represent?

For each domain of interest, compare existing components (both from library and other sources) to the domain models (if they exist). What components are there? What components are missing but are needed?

How are the candidate domains supported?

- No support.
- Reuse library built by extracting from previous work.
- Domain-specific library infrastructure.

Library plans (see Section 3.4.2.1) and library feedback processes (see Section 3.4.2.2) also play an important part in analyzing reusable components and their availability.

3.4.2.1 Library Plans

A reuse library should have an implementation plan that may include:

- Domain requirements characteristics.
- Generic architectures or DSSAs representative of the common solution architectures in the domain.
- List of potential components implementing the DSSAs.
- Component qualification procedures and domain criteria for accepting components into the library.
- Knowledge representation to be used for encoding information. Specific mechanisms for browsing and retrieving information from the library.
- Requirements for library tools, such as those for composing systems/subsystems, for automated qualification of components, for interoperation of the library with other libraries/library mechanisms, etc.
- A TRM for the application domain detailing the interconnections with the hardware and software platform and interface standards between components.

A library implementation plan may serve as an excellent source for identifying or deriving component requirements. The commonality and the variance in the domain requirements and the DSSA should be considered while deriving component requirements. The DSSA provides the framework for components in a domain-specific library.

A typical domain analysis product is a list of potential components implementing portions of the generic architectures. An evaluation of existing products (including COTS and GOTS) may be done to examine their suitability within the application domain. This can serve as a basis for deriving component requirements.

The requirements for components are also influenced by library certification and qualification criteria. These criteria specify requirements having to be met by components for inclusion into the library. The criteria employed by traditional reuse libraries measure a component for modularity, maintainability, adaptability, etc. Additionally, in a domain-specific library, there may be domain criteria to measure a component's "form, fit, and function" within the library. The CARDS component qualification procedures are described in Section 3.6.3.

Knowledge representations and tools used in the library influence the requirements of components. These may be significant factors for reusable requirement, design, document, and test components. For code components, these factors would influence the information needed to catalog the components.

Requirements for components can be derived from the TRM mandated for the library. Typically, the lower layers of the TRM (hardware platform, operating system and communication protocols) are standard across many domains and libraries. Hence, components developed based on a TRM may have wide applicability.

3.4.2.2 Library Feedback

Reuse libraries are continually evolving. Thus, there is often a need for new components within the library. Typically, an operational library collects information on the user's perception of it's contents. This information may be useful in identifying areas the library does not yet provide adequate services. Problem reports on components may also highlight the need for alternate components in the library. Feedback from library usage will become a very important source of requirements when many domains have established libraries and reuse is pervasive throughout government and industry.

3.4.3 Domain Engineer Tasks and Responsibilities

During requirement analysis and specification the domain engineer coordinates a team of component creators and/or SPO engineers. The domain engineer ensures the products of domain analysis are incorporated into component requirements. The domain engineer may in effect act as a customer during this phase and supply lists of mandatory or optional requirements, depending upon the domain analysis.

3.4.4 Component Creator Tasks and Responsibilities

The component creator is usually involved in the creation of requirements depending on the tasks delegated by the domain engineer and the initiative of the component creator. As a minimum, the component creator evaluates the requirements to determine if the creation of the required components is possible with the organization's experience, technology, assets and schedule.

3.4.5 SPO Engineer Tasks and Responsibilities

The SPO engineer may be a prime contributor to the development of requirements because the requirements will effect the systems the SPO engineer is responsible for. The requirement analysis and specification phase of component development may involve a component acquisition RFP. The SPO engineer is a key technical contact for the RFP effort and evaluation of proposals. For details of the SPO engineer's responsibilities during acquisition refer to [CARDSc].

3.4.6 Evaluation

Component design and code is the next phase of component creation after requirement analysis and specification. For the SPO engineer to determine if the requirements analysis and specification is complete, the following goals should be evaluated:

Goal: Confirm that domains have been thoroughly analyzed to derive adequate requirements for reusable components.

Questions:

- Have candidate requirements been checked for commonalities with requirements in government and commercial domains other than the target domain?
- Have existing systems in a domain been thoroughly analyzed?
- Has evolving technology within a domain been considered?
- Are the requirements specified in such a way that they don't define how they should be implemented?
- Are the requirements specified so that they are testable and measurable?

Goal: Confirm that libraries have been thoroughly analyzed to derive component requirements.

Questions:

- Have all reuse libraries been checked to see if the requirements for the candidate components are not already satisfied by existing components?
- Have certification and qualification criteria for target libraries been considered when deriving component requirements?

3.5 Component Design and Code

The design and code process for reusable components takes approximately 60% more effort than the equivalent process for modules intended for one time use in a system [TRACZ93]. The component has to be used about three times to cover this extra cost [TRACZ93]. During the detailed design and coding phase, emphasis is placed on making it easy to implement variations. Some languages, like Ada, make the development of reusable components easier. This section briefly surveys design for reuse and coding reusable components and also explains how to develop and represent reusable architecture and design components and document reusable code components. Also, this section focuses on domain-specific code components and not general-purpose components.

Code components can be developed "from scratch" for reuse in a domain. There are other sources of reusable code components:

- COTS,
- GOTS,
- reengineered code from existing systems.

Tools are available to support reengineering of code into reusable components. When developing new code, tools may not have adequate representations and functionality to support reuse of design and code. Carefully evaluate existing tools and build new tools tailored for the domain. See Chapter 4 for guidance.

3.5.1 COTS Reuse

Current COTS products usually require one or more "wrappers" (software allowing a component to interface with other components). Wrappers are frequently used to interface components written in different languages (e.g., C and Ada) or to convert data into a format defined by the architecture (e.g., SQL). The degree of effort required to integrate a component into an architecture can vary significantly. Some COTS components were developed as stand-alone products without considering external integration issues and therefore may be very difficult to integrate. Some COTS components are designed with certain assumptions involving control flow that make it difficult for it to coordinate with other components. Poor documentation can also greatly hamper the integration of a component. On the other hand, some COTS components are explicitly designed to be integrated (e.g., an SQL database). Also some COTS components are designed as tool kits so they can be configured specifically for integration into a system.

OMG is working on standardizing the interfaces to large objects (i.e., COTS components) within CORBA [OMG92]. OMG has developed an interface definition language (IDL) similar to C++. Bindings to the IDL can be written in other languages (a C binding exists now). The OMG has also defined a Basic Object Adapter providing standard "glue" (i.e., a wrapper) so components can be integrated into a CORBA based heterogeneous system. Special purpose adapters can also be defined. CORBA is still evolving, but if it is widely adopted by vendors, CORBA will greatly facilitate the reuse of COTS components.

There is often no source code available for COTS, so modifications to add required or desired functionality are difficult. In this case, wrappers can be used to augment functionality. However, augmenting functionality with wrappers may not be cost effective when major functionality is missing (e.g., the COTS component doesn't have a graphical user interface or a database interface). In this case a different COTS component should be considered. Another alternative is to combine two or more complementary COTS components together with a wrapper to make up one component in an architecture.

As a result, vendors need to examine groupware (the term is difficult to define, but involves communications technology, object management (management of loosely structured, highly interrelated data), database interoperability, and development tools) as a way of assisting reuse libraries. To increase functionality, and thus reusability, groupware is pushing for open systems.

License costs and maintenance fees are significant issues for COTS components. If numerous copies of the component are needed in deployed systems, a high cost COTS component may not be practical. In this case lower performance/cost COTS and GOTS components should be investigated.

Example: PRISM took a COTS component, DECmessageQ, and added two wrappers to produce a component fitting into the command center architecture. One wrapper written in C directly interfaces with DECmessageQ, and the other wrapper, written in Ada, interfaces between the C wrapper and other components in the architecture.

3.5.2 GOTS Reuse

GOTS components, like COTS components, often require wrappers. Integrating GOTS components presents the same kinds of challenges as integrating COTS components. GOTS components usually come from existing systems in a domain or they may have been developed as a stand-alone system for another government organization. Reusable GOTS tool kits have also been developed. It may be cost effective to build extensive wrappers on a GOTS component to add capabilities, especially for some domain-specific functionality where no COTS component is available (e.g., missile guidance).

Source code should be available for GOTS components, so simple modifications should be practical. These modifications are frequently corrections to problems encountered during integration such as fixing machine and compiler dependencies. However, at some point it may be more cost effective to develop a new component from scratch than it is to modify a component. This is because of the effort necessary to understand how the existing component works internally.

Example: PRISM took a GOTS component, Message Translation and Validation (MTV), added a wrapper and modified the original code to derive a reusable code component. Modifications were not difficult because MTV was originally designed to be reusable. MTV consists of code templates and instructions for template instantiation. CARDS developed a graphical interactive tool for MTV instantiation.

3.5.3 Reengineering

Reengineered code implies major modifications to enhance reusability or modify functionality in contrast to GOTS components which should only require minor modifications. Reengineering code into reusable components is usually done in the context of domain engineering (e.g., collecting an initial set of components) whereas modified GOTS reuse is often done in the context of application engineering (e.g., modify to fit system). Reengineered code can come from an existing system written in Ada or a different programming language. Tools exist supporting the understanding of existing source code and the translation of source code to another language. Reengineering is a good source of reusable components because [FEILER93]:

- legacy code has proven algorithms,
- legacy code can be analyzed to identify good candidate components,
- it is hard to justify completely throwing away legacy code.

Reengineering may not be a good source of reusable components if:

- the leg code is ill structured and poorly documented,
- the legacy code is based on obsolete assumptions.

Extracting reusable components from the legacy code of one or more systems can be divided into a two step process:

1. identify components - There are at least two approaches to identifying components. It is probably best to use a combination of these two approaches [CALDIERA91]:
    - Identify desired functionality from the domain model and use static and dynamic analysis techniques to pinpoint the areas of code where this functionality is implemented. Static analysis techniques would help search for relevant data and operations and visualize the structure of the program. Dynamic analysis techniques would help identify what areas of code get executed based on specifically invoked functionality.
    - Identify components directly from the code. Looking at the modularization of the code gives clues to how designers originally decided to bundle functionality. Looking at how frequently a unit of code is invoked gives clues as to what units are useful and thus potentially reusable.

2. reengineer components - A combination of a number of techniques should be used to enhance and repackage the components identified in the previous step:

   - Modify and restructure the code according to the guidelines for new code (discussed in Section 3.5.4). Static analysis and metrics tools can help identify problem areas.

   - Make the (or at least the externally visible) operation names, variable names, and comments consistent with the terminology used in the domain model.

   - Add black box functional specifications, test cases, classification/context information, and maintenance heuristics to help document the component.

Example: The STARS Army CECOM demonstration project is doing reengineering in the context of reuse. The demonstration project is reengineering a particular Intelligence Electronic Warfare (IEW) system in the context of doing domain engineering in an IEW subdomain across systems. Components developed/reengineered during the demonstration project including requirements, design models, code, and test cases, will be consistent with the domain model being developed concurrently. These components will be used in the system being reengineered as well as for future IEW systems.

3.5.4 Reuse Guidelines for New Code

The following are issues to be considered when developing reusable code "from scratch" or when modifying components. Some of the guidelines apply to software written in Ada only, while other points are generally applicable to many programming languages. See Appendix D for more information about writing reusable source code.

- **Portability** is critical to reuse because components will inevitably be moved to computer systems they were not built on. There are several reports addressing the topic of coding for portability [STARS90][NRL91][SPC92a].

- **Maintainability** is critical for reuse because there will often be a need to understand and modify the code [STARS90]. Examples of tools for checking code maintainability/reusability include:

  - Standards_Checker is a NRaD/Ada tool which checks for the use of standard numeric types, nesting, and the use of "USE" in a context clause.

  - Ada Assured checks code against the SPC Ada guidelines.

  - ANNA is capable of documenting and verifying the constraints and semantics of Ada programs; making it possible to uncover design flaws or inconsistencies.

In addition to portability and maintainability issues, which are key ingredients of quality software in general, there are issues specific to reusability to be considered in the detailed design and coding of Ada components [GAUTIER90][ATKINSON91][COHEN90][HOLLING92].

First, software must be "bundled" or encapsulated into reusable components. Ada packages are the key mechanism for this bundling. Packages allow the grouping of data and procedures into components. However, these components may not be large enough to fill elements of a DSSA for large-grained reuse or megaprogramming. Ada9X will improve support for bundling and provide hierarchical libraries which will allow the grouping of related packages into a subsystem. These subsystems should make better components for large-grained reuse.

Facilitating variation at the code level is another reuse issue. Ada generics facilitate the variation of data types in packages and procedures. A generic is a code template capability built into the language allowing the reuse of algorithms with interchangeable data types. It is difficult to anticipate variations in domain systems. Since new systems often are developed to incorporate the latest technology, the domain requirements may be a constantly moving target. Reusable design and code components should be developed from the start with a plan for their evolution.

Reuse at a higher level of abstraction can be achieved with an object-oriented (OO) approach, i.e., it is easy to create component variations in terms of data (not as limited as generics) and procedures. This is done by inheriting data and procedures from a parent component (i.e., this new component is just like that one except). Ada9X will incorporate inheritance directly into the language. There are two Ada extensions currently supporting inheritance. Both extensions allow someone to program with extended features and then translate to Ada:

- Classic Ada (by Software Productivity Solutions),
- DRAGOON (by the Esprit Project [ATKINSON91]).

Variation can be implemented at an even higher level of abstraction with 4th generation languages (4GLs), application specific languages (ASLs) (see Chapter 4) and graphical design languages (GDLs). 4GLs have been used for some common business applications like spreadsheets with great success. ASLs have been developed and successfully used for specialized tasks in limited domains such as message processing. GDLs have been used in domains where graphic notations are natural (e.g., control systems). 4GLs, ASLs, and GDLs capture the variation at a requirement or design level and automatically generate code. ASLs are strictly text and thus can capitalize on compiler technology, whereas 4GLs and GDLs have a mixture of text and graphics. When a domain has many similar components that can be parameterized, then it is more effective to build an ASL or GDL than to develop and distribute large numbers of component flavors. In some cases there are an infinite number of variations, and ASLs or GDLs are the only practical approach. The ASL or GDL translator/compiler would then be included in a reuse library rather than actual code components. It is likely that a combination of generics, object-oriented approaches, ASLs, and GDLs will be used to develop a set of components covering a domain.

3.5.5 Domain Engineer Tasks and Responsibilities

The domain engineer provides answers to questions generated by the component creator and the SPO engineer during the design and code phase of component creation. The domain engineer is usually a specialist in the domain and is not available for the day-to-day interface and monitoring tasks accomplished by the SPO engineer.

3.5.6 Component Creator Tasks and Responsibilities

The main burden of software component design and code falls on the component creator. The process of developing reusable software begins with the search for reusable components as described earlier in this section, but also includes all of the standard software development practices of peer reviews, walkthroughs, and technical reviews with the customer (domain engineer or SPO engineer). If reuse of software and documentation is used, and the requirements are written with reuse in mind (for example, specifying standard communications protocols, documentation formats, and message formats), then the effort and cost of design, code and documentation can be greatly reduced.

3.5.7 SPO Engineer Tasks and Responsibilities

During the design and code phase of component creation, the SPO engineer is often the interface between the domain engineer and the component creator. Often, issues and questions arise about the domain or requirements that would only be discovered during the design and code phase. During this phase of component development, program progress reviews and technical interchange meetings will include both the SPO engineers and the component creators. It is the responsibility of the SPO engineer to verify that all requirements specified are met by the design and code presented by the component creator.

3.5.8 Evaluation

As part of the software design and code phase of component creation, the following goals should be reviewed by the SPO engineer and the component creator. These goals are specific for software reuse and are new to many organizations; therefore, it is advantageous to review these goals periodically during software design and coding.

Goal: Ensure code components are reusable.

Questions:

- Are the code components portable across:
  - Hardware platforms?

  - Software platforms?

- Are the code components developed according to standards promoting maintainability?

- Have code components been engineered to accommodate variations?

3.6 Component Testing

Reusable components should be tested by the developing organization and then be "certified" or "qualified" by the components' library personnel. This section covers testing by the development organization. Component certification is described briefly (i.e., it is compared to qualification) but is explained in detail in [ASSET92]. This section also covers component qualification for domain-specific libraries based on the CARDS approach. It is important to understand component qualification when developing component requirements (Section 3.3) as well as when planning component development testing (this section).

The following testing types are discussed in this section:

- development level testing of reusable components (Section 3.6.1),

- formal testing (Section 3.6.2),

- component certification and qualification (Section 3.6.3),

- usability testing (Section 3.6.4).

3.6.1 Development Level Testing of Reusable Components

Testing reusable components that will be marketed or distributed is like testing any software product except that there is more of a challenge in determining the variety of contexts in which the component will be used. By developing components to fill parts of a domain-specific architecture, this variety of contexts can be constrained somewhat. If the component is developed for a particular domain-specific architecture then the test is like a "gauge" checking if the component is interchangeable in the domain systems [COX90].

There are several factors driving the development of reusable component tests, such as:

- Test cases and their associated functional requirements help communicate what the component does to potential users.

- Test cases and results may be submitted to a reuse library as proof of quality in a certification or qualification process.

- Test cases may be rerun to certify the quality of the component.
- Test cases may be modified and rerun to test the modified component.
- Test cases should test portability by compiling on one or more operating systems using two or more compilers, and then by executing reliably on two or more operating systems.
- Selected test cases may be extracted and composed into a test for a system that includes the component.

Software tests require more detailed focus if the software is to be subsequently reused. The more widely a software component is used, the more problems latent errors can cause. The cost of error removal increases with reuse. Thorough testing is needed, using both black- and white-box testing approaches [EVB93]. Test efforts should at least include systematic black-box testing at the level of the whole component. However, for large-scale components, which are the subject of this handbook, black- and white-box testing at the unit and integration levels is still necessary and these tests should also be documented for future reuse. The requirements should be defined and documented in some representation that is semiformal (i.e., defined syntax and semantics - but still easy to use), machine processable and reusable. Terminology should be defined and checked for consistency. Input and output data types and formats should be defined in detail. Test cases must be derived from and be traceable to the requirements. These are desirable characteristics of reuse tools for test case generation and requirements analysis mentioned in Section 4.3. Test execution and coverage analysis tools are useful at any testing level.

To help understand the requirements and focus testing resources appropriately, an operational profile should be developed [MUSA87][AGGARWAL93]. For the operational profile, all requirements should be ranked in order of importance based on the following set of factors:

- frequency of function use - what percentage of time is this requirement/function executed out of all the component requirements during average use?
- function criticality - how important is this requirement to the component user?
- function failure criticality - what are the consequences of the failure of this function (e.g., inconvenience, significant financial loss, or death)?
- user criticality - how important is it to satisfy the primary user of this function (e.g., end user versus system administrator)?

The more important the requirement, the more the input space should be sampled for that requirement during black-box testing. The operational profile also gives the potential user of the component background information related to the component's domain and the priorities of the component developer.

3.6.2 Formal Testing

Most government contracts require formal testing, i.e., explicit test documentation with scenarios and test programs, when appropriate, and formal test execution conducted by the component creator (or contractor) but closely monitored and verified by the SPO engineer. Formal test plans and procedures are usually specified in the contract or referred to a military standard. DoD-STD-2167A [DoD88] specifies all document formats for component design, development and test. Some government contracts demand a requirements database with tight controls on modifications to track requirements, where they are satisfied in the code, and where they are tested.

The reuse of formal test plans and procedures (or at least potions of them) would reduce the cost of their development. If the reuse has started at the system design or specification, then reuse of test documentation is much more likely. Test plans tend to be more high-level and may be more reusable than test procedures since they tend to be machine specific.

Tools used during formal testing, such as scenario generators and specific test programs, should be treated as reusable components as well. They should follow the reuse guidelines described in this handbook for component design and code.

3.6.3 Component Certification and Qualification

Component qualification is done by domain engineers who are building a reuse library. The component developers should be aware of the qualification process and account for it in their testing process. For more information about CARDS library development (e.g., library evolution and component qualification) refer to the CARDS *Library Development Handbook* [CARDSe]. For more information about CARDS library operation's policies and procedures refer to [CARDSf and g]. The distinction between component certification and component qualification can sometimes get blurred. For this handbook the following definitions are used:

- **component certification** - The process of determining if a component being considered for inclusion in a library meets the requirements of the library and passes all testing procedures. Evaluation takes place against a common set of criteria (reusability, portability, etc.).

- **component qualification** - The process of determining if a potential component is appropriate to the library and meets all quality requirements. Evaluation takes place against domain criteria.

This section describes the CARDS qualification process which could be adapted by others.

The unique nature and characteristics inherent within a model-based library and the emphasis on domain-specific reuse causes the qualification process to be more focused than the certification process used for other libraries. Also, based upon the CARDS experience with a Command Center Library (CCL), CARDS has shown that the life cycle of the model-based library is iterative and cyclic, with the process of component acquisition and qualification affecting both the modeling and domain engineering tasks.

The CARDS CCL uses a model-based representation paradigm. Model-based libraries support a "components in - systems out" scenario that is consistent with DoD's goal of megaprogramming. To achieve the composition of systems, the CARDS libraries, unlike traditional, component-based libraries, contain more than software modules. Megaprogramming is aided by a library incorporating a mapping of the problem space (the domain) into the solution space (generic architecture). A library model encodes requirements of the domain, a generic architecture, and those software artifacts traditionally comprising software reuse libraries.

The collection of the problem and solution spaces resident within the CARDS library models causes a natural partition of the library model domain criteria into groupings of constraints: component , architectural, and implementation. These partitions (as illustrated in Figure 3-3) correspond to the different by-products produced by domain engineering. The component, architectural, and implementation constraints resident within the CARDS library models form the basis for component qualification. Below is a more detailed description of these major constraints:

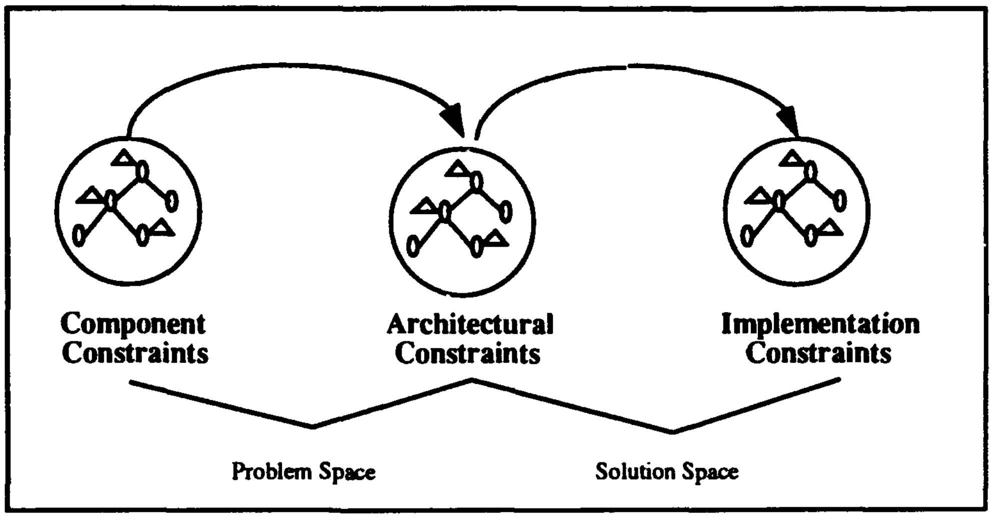

Figure 3-3 Representation of CARDS Command Center Library Model

- **component constraints**. Component constraints represent the mission-level requirements identified within the domain's boundaries. They determine the functionality of the system expressed in terms and language dominant within the domain. They are one of the domain analysis task products. Examples of component or domain constraints for Graphical Information Systems (GIS) in the command center domain are "select map data" and "zoom/pan maps."

- **architectural constraints**. An architecture represents the set of modules or subsystems comprising a completed system and the relationships between cooperating subsystems or modules. Architectural constraints are a formalism of the relationships between these subsystems and any limitations placed upon them. The process of domain engineering, where the generation of a generic architecture occurs, provides the establishment of architectural constraints. Examples of architectural constraints would be specific data flows within the system (e.g., for GIS "output data to SQL database").

- **implementation constraints**. The particular hardware/software environment where the library system resides and is expected to operate provides the basis for the implementation constraints within the model. These constraints provide the requirements the individual components (software modules) must adhere to. Examples include particular hardware systems and operating systems within which components must operate, and types of window systems for components with graphic user interfaces.

The inclusion of the above three major constraints affects the role of component qualification. A significant result is that many of the steps performed for placing software artifacts within a component-based library [RAPID91][ASSET92] are reversed for the model-based CARDS libraries. Within component-based libraries, a candidate component is evaluated based on its reuse potential, modification effort, and general characteristics. Reuse potential is determined by the component's general functionality and its perceived relativity to the library's clientele. The effort to reengineer the component to existing library coding standards is determined. The component's general characteristics include: reliability, maintainability, and portability. After a candidate component successfully passes these evaluation criteria, it is tested and classified. Testing consists of running the component against a test suite or, if none exist, the test suites must be developed. A component is classified to determine its venue within the library.

Model-based library component evaluation, with the emphasis on supporting domain requirements, reverses this process in two significant areas. First, the candidate component is classified as to which subsystem of the generic architecture it satisfies. This classification is not always straightforward because different types or classes of components may fulfill many different areas within the architecture. Depending upon the specificity of the generic architecture, candidate components may or may not be readily derived. The second major difference is that testing is performed before acceptance. Testing evaluates both the candidate component's general performance and its capability of performing the features required of it by the generic architecture.

CARDS views the testing aspect of component qualification as a two-tier effort. The first tier of qualification measures the potential component against domain criteria, the measurement of the "form, fit, and function" applied to a domain. These criteria are a composite of component, architectural, and implementation constraints as described above. Second, a potential component is measured and evaluated according to its general or common characteristics regarding its performance, reliability, maintainability, etc. These characteristics are referred to as common criteria and are not influenced by the domain. They can be thought of as domain independent and include the type of component qualification typically performed by component-based libraries.

The domain criteria are a combination of the component constraints, architectural constraints, and implementation constraints residing within the CARDS library models. An important task in deriving the domain criteria is the decomposition of the domain constraints into the requirements or functionality required by each subsystem of the generic architecture. Generally, these constraints are expressed in domain terminology and not in terms readily transferred or translated into the capabilities of off-the-shelf products. Depending upon the specificity of the domain engineering products and the domain experience of the personnel performing the evaluations, this mapping can be difficult and time consuming. This activity must be performed for every type of component class considered for inclusion into the CARDS library models. More than one class can be used for a single subsystem within the architecture.

The common criteria refer to those characteristics typically performed within component-based libraries. These criteria include determining the component's reliability, maintainability, portability, etc. A key distinction with this CARDS assessment is that most components evaluated for inclusion into the CARDS libraries are typically COTS or GOTS software systems more complex than software modules. Therefore different methods are usually employed for determining if the candidate components meet the common criteria. The use of automated tools are geared towards source code. CARDS relies upon other techniques for determining that information. Such techniques include examining the manuals, determining the hardware/software platforms the component operates on, determination of on-line help and technical support from the supplier, maturity of the component, bug reports, revisions, test procedures supplied by the supplier of the component, independent reviews of the component, etc.

Several factors must be considered when evaluating a candidate component. The determination of whether or not a component meets the domain and common criteria is, in most cases, not a straight-forward task. Questions to be asked include:

- How well does the candidate component meet the domain criteria? Does it meet all of the critical criteria? What are the acceptable variations in performance?
- If some part of the criteria is not met, can wrapper software be developed to makeup the missing functionality?
- If wrappers are required, what is the feasibility in terms of development time and resources?
- Can the candidate component be integrated with other components to form a composed subsystem? Are there any performance degradations after integration? Are wrappers necessary for integration? Are the integration tests performed with a live system or with a prototype and does it matter?
- With the common criteria, does a poor evaluation with regards to portability, reliability, etc., affect or contradict any domain criteria? Will it affect any of the many combinations of composed systems?

CARDS is attempting to remove as much subjectivity as possible in measuring a component against the common and domain criteria. One early established CARDS goal for component qualification was not to have a precise metric (or "magic number") to singularly determine a component's quality for inclusion into a library. CARDS is striving to remove all subjectivity during the qualification process and to still provide meaningful information. Thus, work on further refining component qualification and measurements is continuing.

Test plans or scripts are developed to test each of the domain criteria. These scenarios ensure different components within the same class can be evaluated in the same manner and the evaluation results can be compared to each other. Components are then installed and tested against all criteria. This includes testing component constraints (especially those that are critical), performance issues, usability issues, and integration issues.

The measurement of a candidate component against the domain and common criteria is an important issue within the reuse process. To fully appreciate the role of qualification, it is important to understand the entire library population process. This process consists of three phases:

- **acquisition**. Potential software products are identified, screened and then evaluated. The purpose of product identification is to compile a list of potentially reusable software products suitable for the domain and obtain the necessary information about them to conduct a product screening. The goal of the screening process is to prioritize the list of potential products so more detailed and time consuming evaluations can be performed with a high acceptance rate. After the product passes the screening phase, it then becomes a candidate component for inclusion into the CARDS library and proceeds to the evaluation process, which consists of measuring the candidate against the domain and common criteria. This evaluation leads to a recommendation from the evaluation team whether or not to accept the product and include it in the library. Products best suited for the domain-specific reuse library are then integrated into the library. Usually the products require enhancements and/or modifications. When this occurs, required adaptations are specified and then sent on to the adaptation phase.

- **adaptation**. Existing software components are modified or enhanced (i.e., wrappers) as new software components. During the adaptation phase, software components are designed, implemented (coded) and then tested in a stand-alone configuration. The results are fed back to the acquisition phase.

- **integration**. Library products, regardless of their origin (i.e., off-the-shelf, modified-off-the-shelf, and wrappers), need to become an integrated part of the library. Much of the work performed in this phase is directed at validating the integration of a component into the library as well as with interfacing components as defined by the library model. Depending on the nature of the component being integrated and the interfacing components (i.e., is there executable code present for the component?) the integration can consist of analyses and tests and produces integration procedures and reports.

Typically, library creation is a linear process. However, the CARDS library model creation/development (see Figure 3-4) is an iterative on[illegible] area is component qualification which provides several functions affecting the process [illegible] library model creation. As previously stated, the key role of component qualification is to measure the component's "form, fit, and function" against the constraints inherent within the library.

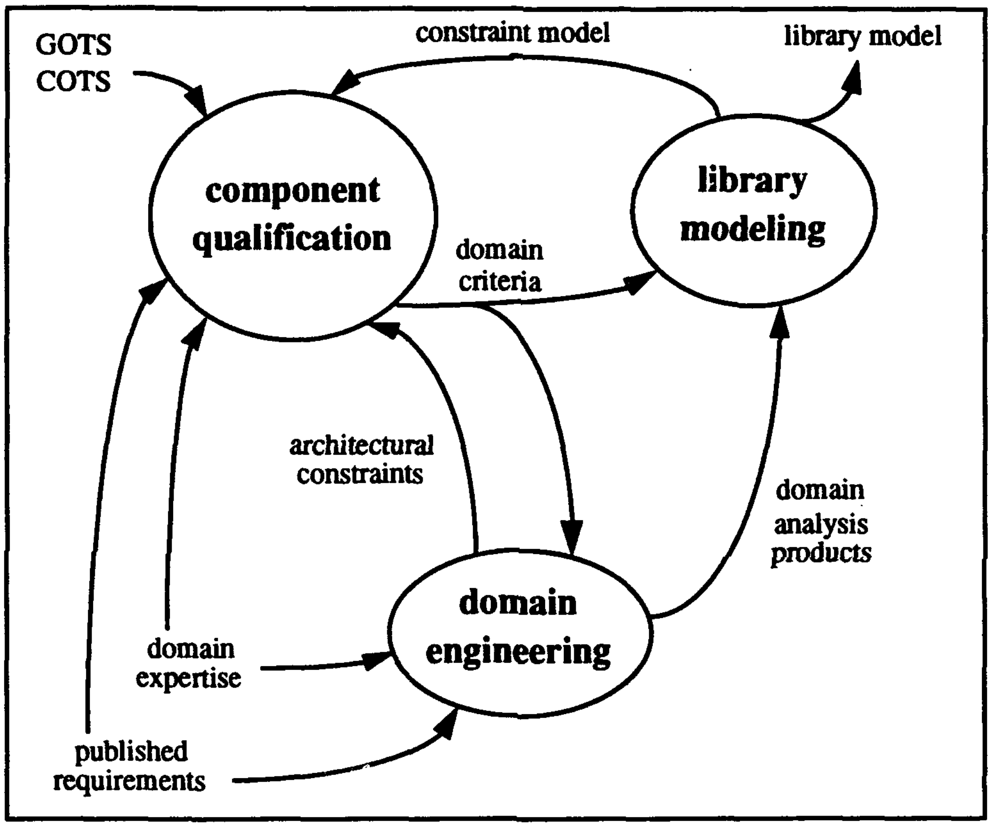

Figure 3-4 Feedback and Iteration in the Model Development Process

The act of measuring/evaluating components leads to a few insights into the library model:

- It provides a basis for determining the appropriateness or correctness of the architecture. An overly restrictive architecture can be discovered by difficulty in finding/obtaining components suitable for inclusion matching the constraints of the architecture and the desired functionality. Another indication is the excessive need for developing wrappers used to provide missing functionality from the original component. This information may uncover that the architects did not thoroughly understand or were unaware of the appropriate software for populating the model-based library. Architecture modification may then be required to make it more responsive to the existing software base. This is important for systems wanting to leverage reuse of existing software for library population.

- The component qualification process is closer to the constantly evolving nature of the software industry and software developers. This closeness manifests itself in realizing that technological change within software may render pieces of the architecture unnecessary or several modules within the architecture may be performed by single products that would reflect combining those modules into a single module. Technical evolution may also force the alteration of relationships between architectural modules. In cases where the architecture is not altered, the architects should at least reexamine it for relevancy.

- Component qualification produces the domain criteria (i.e., forms of constraints). These constraints are fed into the library model, through the domain engineering process. A refinement of these criteria is constantly being performed and the results are fed-back into the library.

3.6.4 Usability Testing

Usability testing is the objective evaluation of the "user-friendliness" of a software application or component. Since reusable components will, hopefully, be reused many times, it is important to include usability testing as part of the reuse components' test suite to reduce the cost to the program of future costly user interface modifications. When two COTS components have equivalent functionality, usability becomes the discriminator for choosing one component over another. [ABELOW93] *highlights the importance of usability testing:*

"Usability testing is being implemented to ensure accuracy, especially with user interfaces. If you nail your software's user interface, you'll increase it's quality, acceptance, and revenues, while reducing future costs for customer support and maintenance".

Jacob Nielsen lists the following features of a usable system [NIELSEN93a]:

- easy to learn, so new users can quickly start doing some work,
- efficient, so the expert user can take shortcuts,
- easy to remember, so infrequent users can easily use the system,
- relatively error-free or error-forgiving,
- predicts user error or recovers easily from them,
- pleasant to use.

The inclusion of usability testing in test suites for software is a relatively recent event and has had the added difficulty that much of usability testing could be considered subjective and therefore hard to quantify and qualify. Jacob Nielsen [NIELSEN93b] lists the following benefits resulting from usability testing:

- increased user productivity through improved interface design,
- decreased user training time (a onetime costs saving),
- increased satisfaction,
- reduced support costs (such as staffing for support hotlines),
- increased sales,
- if applicable, improved reviews in trade magazines resulting in free marketing and good press.

The exact usability testing financial benefits are hard to measure. [NIELSEN93b] includes some guidelines of how to measure the financial benefits associated with usability testing. Although usability testing is a relatively new science, some methods have been developed to provide guidance for this form of testing:

- Usability testing is an iterative process and should be tested at every new release of the interface.
- The system should be evaluated for tasks that are routinely executed and timed at each iteration of the testing.
- Usability testing should be executed with test users who most closely represent the actual system end users. Iterative usability tests should use a new set of users for each iteration to eliminate transfer of knowledge from the test results.
- User error counts and their impact on the system should be measured with each iteration of usability testing. The goal being to eliminate errors a user can make with the user interface.
- Subjective tests such as "user-friendliness" should be defined objectively for frequently executed tasks in terms of increased system and user performance.
- Iterative usability testing should be included in the test plans and scheduled for all development efforts. Usability testing must therefore also be included in the test requirements specifications.

[NIELSEN93a] and [KARAT92] include information on actual usability testing case studies. [KARAT92] also includes information about empirical testing versus walkthrough testing methods for these case studies.

3.6.5 Domain Engineer Tasks and Responsibilities

Domain engineering tasks and responsibilities during the test phase of component creation are very similar to design and code responsibilities. Domain engineers provide answers to questions generated by the component creator and the SPO engineer. Domain engineer are usually domain specialists and not available for day-to-day interface and monitoring tasks accomplished by the SPO engineer. Domain engineers may be called upon to review drafts of test plans and procedures or to review requirement allocation to test levels (i.e., system, software, hardware, or code walkthrough).

In addition to the task of testing systems for compliance to requirements, there is an added step of component qualification and certification with regards to reuse libraries. Component qualification and certification is usually the responsibility of reuse library management which may be a task assigned to domain engineers.

3.6.6 Component Creator Tasks and Responsibilities

The component creator (or other personnel in a separate test organization) is usually obligated to write test plans and procedures, and dry-run and perform formal tests with government witnesses. It is to the benefit of the component creator to execute internal development level tests to provide early detection of any deviation of code from the formal requirements. Numerous studies have outlined the huge financial costs associated with late discovery of program code errors. Usability testing is a relatively new kind of test that may not be contractually required and is often not included in the plans and schedule. The benefits of usability testing have been outlined in this section and it is to the benefit of both the government domain and SPO engineers to include this type of testing in the requirements specification.

3.6.7 SPO Engineer Tasks and Responsibilities

During the component creation test phase, the SPO engineer is responsible for verifying that all requirements are tested and proved operational. In some instances, interpretation of a requirement becomes an issue during the test phase even though it may have been agreed to in the development phase. This ambiguity comes from the fact that many organizations use different personnel to perform testing versus component creation and these groups have very different ways of looking at a system. Most of these errors can be caught early by the SPO engineer by carefully monitoring the test plans and procedures as they are developed. Another responsibility of the SPO engineer during the test phase is the sign-off of requirements as tested and contractually complete. A method and formal procedure for requirement sign-of[illegible]ld be agreed upon by the SPO engineer and the component creator before the start of the [illegible]ase.

3.6.8 Evaluation

Before component testing in any form can be considered complete, the following goals must be evaluated for success:

Goal: Ensure development of adequate reusable component test cases.

Questions:

- Have the component requirements been reviewed thoroughly for consistency of terminology and to ensure they are not ambiguous?
- Have input and output data been specified in detail?
- Has a realistic operational profile been developed?
- Have the test cases been peer reviewed?
- Do the test cases adequately cover the requirements and reflect the operational profile?

Goal: Ensure adequate completion of component testing.

Questions:

- Have the test cases been executed?
- Have anomalies been recorded, fixed and retested?
- Have the results of the test been thoroughly reviewed?

3.7 Component Maintenance

Component maintenance is included in this handbook as a component creation follow-on since, in the area of software reuse, it falls in the category of reuse library management. Component maintenance can result from two types of impetus: routine maintenance or user/designer requests for modifications.

The original component creator may not be part of the team accomplishing component maintenance, but the domain and SPO engineers are certainly involved. The domain engineer is involved in the maintenance of components by evaluating requests for changes to the components. The SPO engineer is involved in component maintenance whenever a change to the component affects the systems that he/she is responsible for. For more information about library maintenance procedures, refer to CARDS *Library Development Handbook* [CARDSe], *Engineer's Handbook* [CARDSc], and *Library Operation Policies and Procedures* [CARDSf and g].

4 REUSE TOOL DEVELOPMENT

4.1 Introduction

Tools help increase human engineer productivity and consistency in the software process. Tools for the support of reuse processes are still in the "Bronze Age." A hodgepodge of tools exist to do the simpler tasks with a large amount of human involvement. Tools taking over the bulk of the decision making from the human will not be available in the near future. The goal of this section is to discuss the evaluation of available tools supporting reuse and to promote the efficient development of new high quality tools to fill the current gaps.

This section covers anything providing automated support to reuse, including tools helping to build systems from components, and tools that build, analyze and evaluate component documentation/code. The scope is narrowed to library-assisted architecture based reuse. This section does not address transformational systems and other more experimental systems. It also does not address generic tools applied to reuse management (e.g., planning and scheduling). Figure 4-1 summarizes the topics discussed in this chapter.

4.1.1 Audience

The intended audience for this section is the developer of reuse tools or the developer of reusable components that has determined a need for a reuse tool. The audiences described earlier in this document (the domain engineer, the component creator, and the SPO engineer) may all find a need for a reuse tool. This section assists these audiences in the determination, selection, specification, and implementation of reuse tools.

4.1.2 Tasks and Responsibilities

There are two reasons for reuse tool development: resulting from a need determined by government domain analysis, or resulting from a need determined by component creators. In the first case, the tasks and responsibilities for the audiences of this handbook are the same as those defined for component development (Chapter 3). When the need for a reuse tool is determined by the component creator and required for internal use (or for a possible new market) the tasks and responsibilities fall entirely on the component creator and support staff (requirement analysis, domain analysis, and testing are accomplished by the development organization).

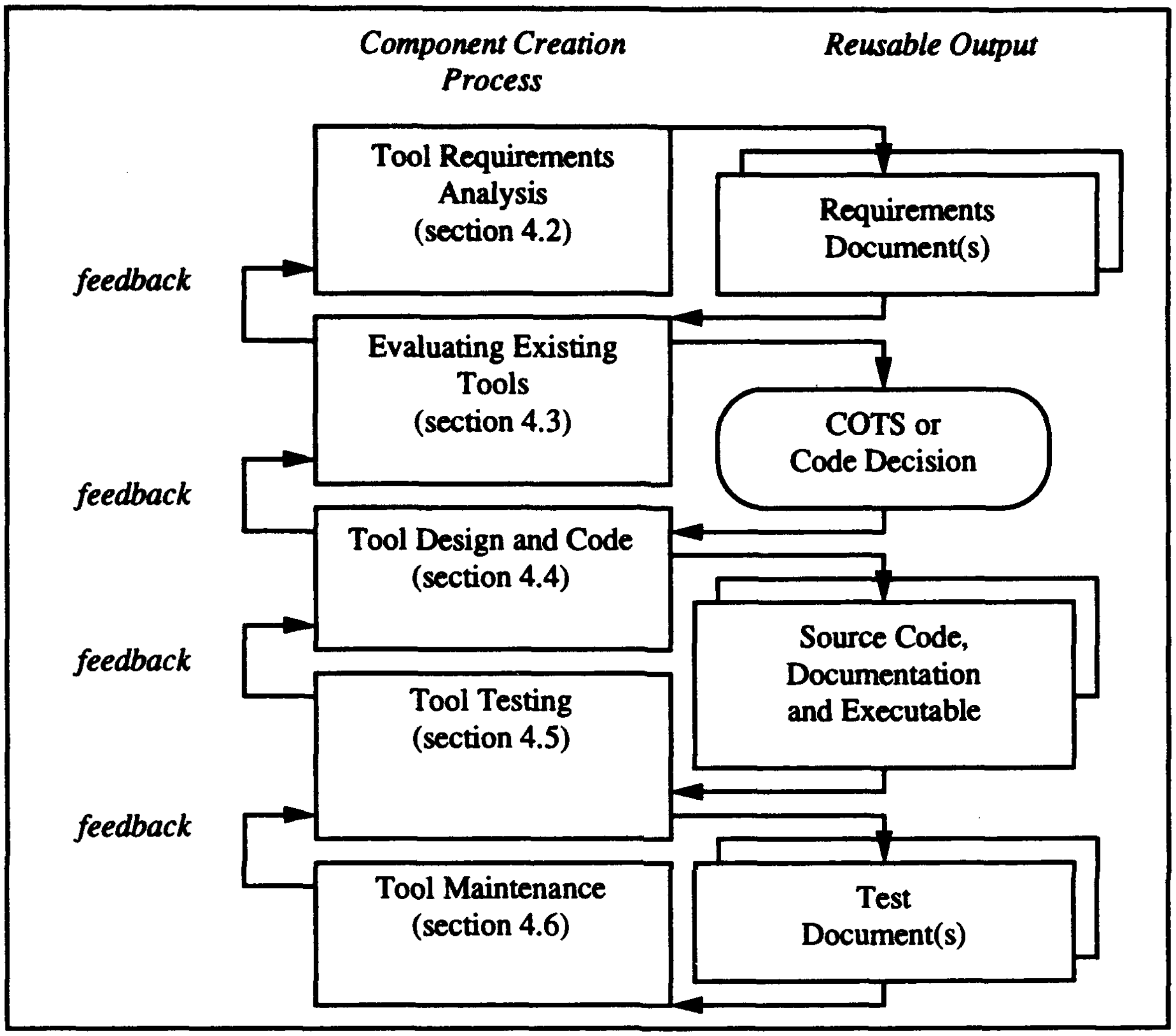
Component Creation Process
Reusable Output
Tool Requirements Analysis (section 4.2)
Requirements Document(s)
feedback
Evaluating Existing Tools (section 4.3)
COTS or Code Decision
feedback
Tool Design and Code (section 4.4)
Source Code, Documentation and Executable
feedback
Tool Testing (section 4.5)
feedback
Tool Maintenance (section 4.6)
Test Document(s)

Figure 4-1 Reuse Tool Evaluation and Development Process

4.1.3 Overview of Reuse Tools

The following is a tools taxonomy for three distinct but related reuse processes. Note that "asset" and "component" are used interchangeably since STARS and CARDS use these different terms. Table 4-1 shows a taxonomy for the development of systems using a reuse process as described in the CARDS *Engineer's Handbook* [CARDSc]. System development corresponds to asset utilization in the STARS *Conceptual Framework for Reuse Processes* (CFRP) [STARS92]. Table 4-2 shows a taxonomy for the process of developing domain-specific components as described in Chapter 3 of this handbook. Component development corresponds to the CFRP's asset creation. Table 4-3 shows a taxonomy for a reuse library development and evolution process as described in Chapter 3 of this handbook (domain analysis, modeling and component qualification), and in the CARDS *Library Development Handbook* [CARDSe] and *Library Operations Policies and Procedures* [CARDSf and g]. Library development and evolution corresponds mainly to the CFRP's asset management. Since there is commonality between activities in these three processes, there is also commonality in the tools used.

Table 4-1 System Development Process

Tool Type	Requirements Analysis	High-Level Design	Detailed Design	Code	Test	Maintenance
Requirements Analysis and Design	X	X	X			X
Programming Support				X	X	X
Code Generators			X	X		X
Documentation	X	X	X	X	X	X
Reengineering	X	X	X	X		X
Static Analyzers				X	X	X
Test Tools					X	X
Application Specific Language Translators	X			X		
Modeling	X	X	X	X	X	X
Component Management	X	X	X	X	X	X

Table 4-2 Component Development Process.

Tool Type	Domain Analysis and Modeling	Requirements Analysis	Design	Code	Test
Requirements Analysis and Design	X	X	X		
Programming Support Environment				X	X
Code Generators			X	X	
Documentation	X	X	X	X	X
Reengineering	X	X	X	X	
Static Analyzers			X	X	
Test Tools					X
Application Specific Language Translators (1)					
Modeling	X	X	X		
Component Management	X	X			

[1] May be built during component development process.

Table 4-3 Library Development and Evolution Process.

Tool Type	Domain Analysis and Modeling	Component Qualification	Library Evolution
Requirements Analysis and Design	X	X	
Programming Support Environment		X	X
Code Generators	X	X	X
Documentation	X	X	X
Reengineering	X		
Static Analyzers		X	
Test Tools		X	X
Application Specific Language Translators			
Modeling	X	X	X
Component Management	X	X	X

Some tool types include general Computer Aided Software Engineering (CASE) while other tool types include many tools developed specifically for reuse. In many cases it is not clear how to classify a tool according to tool type. Some tools do not fit well into any of the categories because they combine the functionality of several tool types. Some integrated tools have the functionality of several tool types. Therefore tool types represent a set of tool functions. The following is a brief description of some tool types:

- **Application Specific Language Translators (ASL)** - tools custom developed to generate code from a high-level specification in a very narrow domain. These are appropriate for situations where there are many variations of similar requirements. Instead of storing numerous "flavors" of a component, one ASL translator could generate the appropriate "flavor" on demand.

- **code generators** - tools generating code from requirements and design information are sometimes called "back end CASE" or "lower CASE", i.e., COTS tools specific to a broad domain.

- **component management** - tools supporting a reuse library allowing the user to browse, retrieve, and examine components. These include RLF, KAPTUR, InQuisiX, CARDS system composition, and qualification tools.

- **documentation** - tools helping produce text and graphics.

- **modeling** - tools facilitating the development of knowledge representations for domains. These include artificial intelligence tools (e.g., RLF) as well as some of the more powerful requirements analysis and design type tools (e.g., Statemate) adaptable for domain modeling.

- **programming support environment** - tools supporting the development of code including compilers, debuggers, configuration management, etc.
- **reengineering** - tools taking source code as input and facilitating the modification of design and translation to another programming language.
- **requirements analysis and design** - tools supporting the early software life-cycle phases are sometimes called "front end CASE" or "upper CASE", e.g., Software Thru Pictures, Teamwork, and ReQuire. These tools generally support one or more requirement or design methodologies.
- **static analyzers** - tools analyzing source code to determine structure, data flows, complexity, and conformance to standards, e.g., AdaMat and Logiscope.
- **test tools** - tools supporting test planning, test case generation, test execution, regression testing, and test coverage measurement.

Foundation technologies underlying reuse tools includes:

- **compilers** - parses and translates code and other forms of information.
- **databases** - stores and retrieves software related data.
- **graphics** - displays software knowledge in the form of diagrams and models.
- **knowledge-based inference** - processes reuse knowledge to achieve a goal.
- **knowledge representation** - captures requirements, design, code, and test information in machine processable form.
- **machine learning** - acquires knowledge in some advanced tools.
- **user-interfaces** - facilitates control and monitoring of tool actions.

The following sections discuss how to find or develop the right tools for system development, component development, and library development and evolution.

4.2 Tool Requirements Analysis

Once a general need for a reuse tool has been determined, a requirements document or list should be developed. This requirements document may be formal (as in Chapter 3 for component development) or informal if the reuse tool will only be used internally. When the requirements analysis is complete an evaluation of existing tools should be performed. If no tools exist or if no tools can be easily modified to automate a particular reuse task, then building a new tool should be considered. This section is intended to provide guidance for defining the requirements for a new reuse tool for internal use as well as for sale outside of a company.

The need for a reuse tool may become evident in a project's planning stages, during the project, or possibly from a perceived market opportunity. Reuse tools are a relatively new domain; however, builders of these tools can draw on CASE experience base and programming support environment tools. Beyond this experience base, engineers are beginning to see artificial intelligence techniques being introduced to support tasks that never had automated support.

4.2.1 Functionality

Carefully choose an area where a tool can automate reuse tasks. In general, routine, repetitious or predictable and commonly done tasks are completed by a variety of technical personnel and are good candidates for tool support. A cost benefit analysis should be done on the identified candidates. The cost for developing, maintaining, and training must be weighed against the improvements in productivity and quality provided by the tool. An existing tool may need to be ported to a new platform or to be significantly enhanced. In this case, requirements analysis is simplified because there is no need to start from scratch.

Knowledge based system tools should be considered for reuse-related decision making tasks requiring short supply expertise and where the decision quality and consistency is poor in practice. However, no tool can be considered a panacea for any type of complex task.

The domain metrics listed in Section 4.3.1 provide a basis for defining requirements for new reuse tools as well as for evaluating existing tools. This list is not inclusive and will evolve along with the technology. The component constraints deal mainly with the reuse tasks that can be automated. The architectural constraints define how the tool integrates with other reuse tools. The implementation constraints deal primarily with hardware and software platform issues. To develop requirements:

- Look at the reuse tasks the tool is planned to support.
- Choose applicable domain metrics (i.e., [illegible]nent, architectural and implementation constraints) as requirer[illegible]or the new tool.
- Elaborate on the requirements to specify more precisely the inputs and outputs of the tool.

4.2.2 Knowledge Representation

The representation of input, output, and internal knowledge/data is very important for reuse tools because reuse depends on accessing and transforming software engineering knowledge. The knowledge representations should be defined early in the requirements analysis process it supports the functionality. Consider these factors in defining the knowledge representation:

- **generality** - must minimize the dependence of the tool on idiosyncratic domain knowledge.
- **levels of abstraction** - must be appropriate for all the tasks performed by the tool.
- **acquisition** - it must be possible to gather the knowledge required for the tool with a reasonable amount of effort and efficiently translate the raw data into processable representations.

4.2.3 User Interfaces

Because there is a person "in the loop" on most reuse tasks, user interfaces are critical to the success of a reuse tool. User interface requirements must be considered in detail early on. The following is a list of general principles forming the basis of specifying and designing the user interface [DUMAS88]:

- Put the user in control (make the response to user commands predictable and make it easy to decide what to do next).
- Take into account the user's level of skill and experience - in the near term most reuse tools will be for software engineers although the goal in the long term is to let the end user build systems with reuse tools.
- Be consistent in terminology, graphics, and "look and feel".
- Protect the user from the inner workings of the hardware and software.
- Provide on-line help.
- Minimize what the user has to remember to use the tool.
- Layouts of information on the screen should promote easy scanning and recognition.

As discussed in Chapter 3, it is important to include usability testing in the requirements specifications. User interfaces for reuse tools should follow the same guidelines and recommendations described for reusable components.

4.2.4 Tool Integration

The area of integrating CASE tools in general and reuse tools specifically is immature. Standards and implementations of standards are just emerging. Tool integration has three main dimensions:

- Control integration involves the sending and receiving of messages from one tool to another to coordinate the tasks done by each tool.

- Data integration involves sharing and passing data from tool to tool. This may be done directly or through a "repository" or an "object management system".

- Presentation integration involves enforcing consistency between the user interfaces of tools or even having the tools share the same central user interface. An integrated reuse tool set is a desirable goal because it will increase productivity and quality. The SEI has done a lot of work in the area of tool integration [BROWN93a][BROWN93b][BROWN94][LONG93].

4.2.5 Evaluation

Once the requirements have been defined, the next phase in reuse tool development is the evaluation of existing tools. Since a COTS or GOTS tool would be the desired tool choice (requiring the least cost due to coding and maintenance), requirements should be specified using standards or de facto standards whenever possible.

Goal: Ensure adequate tool requirements have been specified.

Questions:

- Is the manual process, being replaced by the reuse tool, well defined and documented?

- Has a thorough peer review been done on the requirements?

- Does the tool specified fit the reuse process as originally planned?

- Has a cost/benefit analysis been performed?

- Are the requirements identified as optional/mandatory or given a priority when appropriate?

4.3 Evaluating Existing Tools

Before building new tools, software engineers look for existing tools to be used as-is or modified. This section discusses methods for identifying and evaluating tools. Reuse tools are a subset of CASE tools. "CASE tools are capital equipment for the software factory. They should be selected and purchased with the same care given to purchases of manufacturing machines." [DIXON92] Here are some tool information sources:

- **Ada Joint Program Office/Ada Information Clearinghouse** - provides an on-line products and tools database containing general-purpose tools which may support reuse. **Contact**: Ada Information Clearinghouse c/o IIT Research Institute 4600 Forbes Boulevard Lanham, MD 20706. 1-800-AdaIC-11.

- **Software Technology Support Center** - produces reports on CASE tools which may be applicable to reuse. **Contact**: STSC Customer Service Ogden ALC/TISE Hill AFB, Utah 84056. 801-777-7703.

- **STARS/ASSET** - contains a variety of reports and user manuals for reuse related tools. **Contact**: ASSET 2611 Cranberry Square Morgantown, WV 26505. 304-594-3954 librarian@source.asset.com.

4.3.1 Domain Metrics for Reuse Tools

Once candidate tools are identified, the process of evaluating them is equivalent to component qualification for the reuse tools domain. An overview of the reuse tools' domain is given in the introduction to Chapter 4. The general component qualification process is described in Section 3.5. The following is a collection of reuse tools domain metrics. Many of these metrics apply to software engineering tools in general [IEEE STD 1175][STSC92][SEI87][IEEE92]. Because of the variety of reuse tools, only a subset of these metrics are applicable to a particular evaluation. Therefore, choosing the critical and applicable metrics is an extra step at the beginning of the component qualification process in the reuse tool domain. The common metrics for component qualification are all applicable to reuse tools in the same way they are for any reusable component. The following constraints are discussed in this section:

- component (Section 4.3.1.1),
- architectural (Section 4.3.1.2),
- implementation (Section 4.3.1.3).

The constraints listed in these sections are not intended to be complete but rather to be used as a starting point depending upon the particular tool requirements.

4.3.1.1 Component Constraints

1. Process constraints:
    - Does the tool fit into the current or planned reuse process?
        - Is the reuse process defined and documented?
        - Are there standards for the reuse products?
        - Does the tool produce reuse products conforming to the standards?
        - Can the effectiveness of the tool for improving the process be measured?

- Does the tool support the desired/applicable reuse process/subprocess?
    - System development?
        - Requirements analysis?
        - High level design?
        - Detailed design?
        - Code?
        - Test?
        - Maintenance?
    - Component Development?
        - Domain analysis and modeling?
        - Requirements analysis?
        - Design?
        - Code?
        - Test?
    - Library development and operation?
        - Domain analysis and modeling?
        - Component qualification?
        - Library maintenance?
    - Does the tool fit the role or job function of the intended users?
        - Software engineer?
        - Domain engineer?
        - Program manager?

    - Tester?
    - Quality assurance?
    - Configuration management?
- Is the tool easy to learn?
    - Does the tool provide a tutorial?
    - Is adequate training provided by the tool supplier?
- Is the tool easy to use?
    - Does the tool provide on-line help?
    - Does the tool have an adequate user's manual covering the basic concepts, detailed operations, and troubleshooting tips?
    - Does the tool simplify the task rather than complicate it?
    - Is the tool clear and consistent in its use of terminology?
    - Is the tool clear and consistent in its use of graphic symbols?
    - Is the tool consistent in the "look and feel" of its user interface?
    - Can/does the tool display available options depending on the current state?
    - Does the tool check if the input is valid?
    - Does the tool provide useful error messages?
    - Can the tool be tailored to suit individual preferences and organization standards?
    - Does the tool allow repeated command sequences to be grouped in macros?
- Can multiple users simultaneously use the tool?
- Can multiple users simultaneously prepare the same reuse product?

- Does the tool have acceptable average case response times?

2. Model constraints:

- Does the tool provide adequate representations for the knowledge in the domain (e.g., if the domain has complex mathematics can equations be represented?)
- Does the tool facilitate the creation/editing of domain analysis product representations?
  - Entities, concepts, and categories?
  - Data - variables and constants?
  - Functions, services, and features?
  - Relationships, e.g., specialization and aggregation?
  - Constraints?
  - Vocabulary/terminology?
- Does the tool facilitate the creation/editing of representations for design information and software architectures?
  - Data flow diagrams?
  - Flow charts/control flow diagrams?
  - Structure charts?
  - Object-oriented diagrams?
  - State diagrams?
  - Process timing models?
  - Memory allocations?
  - Algorithms/program design language?
- Does the tool provide consistency and completeness checking on the domain knowledge representations in the model?

- Does the tool allow graphical viewing/browsing of the model?
- Does the tool facilitate the analysis of domain alternatives and variations?
- Does the tool present multiple views of the model tailored for the tasks of specific classes of users?
- Does the tool allow extraction of knowledge in the model for use by other tools?

3. Code constraints:
    - Does the tool support generation of source code from high-level design abstractions or specifications?
    - Does the tool generate reports/diagrams facilitating the understanding of reuse source code?
    - Does the tool provide analyses and metrics to help determine source code reusability?
    - Does the tool facilitate code modification and restructuring?
    - Does the tool translate source code from one programming language to another?
4. Test constraints:
    - Does the tool support test case generation for:
        - Reusable components?
        - Systems built from reusable components?
    - Does the tool support test execution for:
        - Reusable components?
        - Systems built from reusable components?
    - Does the tool support test coverage analysis for:
        - Reusable components?
        - Systems built from reusable components?

5. Component management constraints:
   - Does the tool provide capabilities for:
     - Storing requirements components?
     - Storing design components?
     - Storing code components?
     - Storing test components?
     - Searching a reuse library?
     - Retrieving reusable components from a reuse library?
     - Configuration management/version control of reusable components?
     - Cataloging/indexing reusable components?
     - Collecting metrics on queries to the library?
     - Collecting metrics on usage of reusable components?
   - Does the tool provide a method for linking/tracing requirements, design, code, and test cases?
   - Is the tool compatible with a reuse library storage mechanism based on:
     - An SQL database?
     - An Information Resource Dictionary System (IRDS) database?
     - An object-oriented database?
     - A semantic network?
     - A frame based system?
6. Documentation constraints:
   - Does the tool allow editing and viewing of text descriptions of:
     - Parts of a domain-specific software architecture?

- Reusable components?
- Design rationales and tradeoffs?

- Does the tool support the generation/compilation of documents?
- Does the tool have hypertext capabilities?
- Does the tool support parsing/keyword search of text?

7. Text constraints:

- Does the tool provide storage capabilities for the following formats:
  - ASCII?
  - Postscript?
  - Binary?
  - SGML?
- Can the tool generate:
  - ASCII output?
  - Postscript output?

8. Graphics constraints:

- Can the tool import:
  - PICT and PICT2 vector graphics format?
  - EPS (Encapsulated Postscript) format?
  - TIFF (Tagged Image File Format)?
  - GIF (Graphics Image Format)?
  - PCX graphics format?
  - CGM (Computer Graphics Metafile) format?

  - BMP (bitmap) graphics format?
  - PBM (Portable Bitmap) graphics format?
  - XWD (X Windows Dump) graphics format?
  - PCL graphics format?
  - GKS (Graphical Kernel System) format?
  - PHIGS (Programmer's Hierarchical Interactive Graphics System) format?
  - IGES (Initial Graphic Exchange Specification) format?
  - CDIF (CASE Data Interchange Format)?
- Can the tool export:
  - PICT and PICT2 vector graphics format?
  - EPS format?
  - TIFF?
  - GIF?
  - PCX format?
  - CGM format?
  - BMP format?
  - PBM format?
  - XWD format?
  - PCL format?
  - GKS format?
  - PHIGS format?
  - IGES format?

    - CDIF?
  - Does the tool create/edit color textual displays?
  - Does the tool create paper copies of developed displays?

4.3.1.2 Architectural Constraints

Tool integration constraints (also see Section 4.2.4):

- Can the tool accept or transfer data directly to other applicable reuse tools via:
  - Sockets?
  - Proprietary transfer vehicle?
  - Other IPC vehicles?
- Can the tool accept or transfer data indirectly to other applicable reuse tools via:
  - Text Files?
  - Files of standard format?
- Does the tool provide control integration with:
  - HP Software Workbench Broadcast Message Server?
  - IBM SDE WorkBench 6000?
  - DEC COHESION?
  - Sun ToolTalk?
  - ANSI X3H6 (future)?
- Does the tool provide data integration with:
  - Emeraude PCTE (Portable Common Tools Environment)?
  - ECMA PCTE (future) (European Computer Manufacturers Association) (Portable Common Tools Environment)?
  - IEEE Std 1175 (Trial use Standard Reference Model for Tool Interconnections)?

- IBM AD/Cycle (Application Development Cycle)?
- ASIS (future) (Ada Semantic Interface Specification)?
- CAIS-A (Common APSE Interface Set)?

4.3.1.3 Implementation Constraints

Does the tool have any implementation constraints, such as:

- Maximum number of users?
- Is a pointing device (mouse or pen) required?
- Does the tool run on reuse environment operating system?
- Does the tool run under reuse environment window system or manager?
- Does the tool run on reuse environment hardware architecture?

4.3.2 Evaluation

After an evaluation of existing reuse tools has been completed, a decision must be made to either use COTS software or develop the reuse tool from scratch.

Goal: Confirm that an adequate evaluation of available tools has been done.

Questions:

- Have all known sources of tools been checked?
- Has the tool been evaluated against all critical and applicable metrics?
- Have the tool's capabilities been confirmed by "hands on" demonstration?
- Does the tool evaluation/qualification report summarize strengths and weaknesses?
- Does the tool evaluation/qualification report justify to accept or reject?

4.4 Tool Design and Code

A desirable reuse tool would be generic enough to operate in a wide variety of models and on several types of components within the models. A desirable reuse tool would also be specific enough to handle the peculiar details, the discriminators, making one individual component in a class of components different from another.

It has been CARDS experience that a tool purposely limited in its scope and function, yet assists in reuse, is often the most reusable tool. It can be altered to produce a new tool (e.g., CARDS "Qualification Tool" reuses much of CARDS "system composition tool"). It can also be strung together with other small tools to handle increasingly complex tasks.

This section discusses recommendations for developing successful reuse tools by outlining examples from the CARDS experience in building the model-based system composition and qualification tools. These tools are knowledge based decision making tools. However, many of the ideas presented should be applicable to other reuse tool designs such as graphical and parser type tools.

4.4.1 Tool Design Guidelines

Depending upon the requirements placed on the development of a reuse tool, the design may be as complete as that defined for reusable components in Chapter 3, or as generic as the following sections imply.

4.4.1.1 Help the user

Give users as much guidance/help/explanation as possible as to the process(es) and choices the tool presents. Reuse models can be highly involved; the end user may very well be unaware of the implications entailed by choosing any one of the options presented.

4.4.1.2 Avoid repetitive questions

Forcing the user to answer the same question repeatedly is annoying and should be avoided.

Example: CARDS libraries use the STARS RLF as its library mechanism. RLF library models are semantic networks representing multiple allowable relationships between library assets. The CARDS system composition tool, which is integrated into the RLF, traverses the semantic network and requires the user to specify which of the allowable relationships are required by the newly composed system. A generic version of the tool, running in any area of the semantic network, requires the user to answer questions about non-system composition relationships between assets and, in some cases, questions about a pertinent relationship more than once in the same session.

To eliminate the unnecessary and repetitious questions and the confusion they cause, the system composition tool can be customized with a feature to constrain the tool from asking questions about unrelated relationships and prevent duplicate questions. This feature is implemented using a 'constraint list' attached to nodes of the semantic network.

4.4.1.3 Ensure consistent terminology

Ensure consistent terminology, presentation style, and patterns of doing things. An effort should be made to make the tool's user interface "look and feel" reasonably uniform across the entire tool. This may seem too obvious to mention. For example, clicking on a button in the upper left corner of a window should evoke the same reaction on every window in the tool. Sometimes these interface details are not so obvious. The format/typeface/font size and coloring of tool windows should be consistent. The windows running tools should either be automatically destroyed by the software or should require the user to destroy the window; use the term "exit" or "quit", but not both. If "none" is a recurring choice, place it consistently first or consistently last. Consistency should be evident in the source code as well. The maintainer should receive code appearing to have been written by a single person using a single style.

4.4.1.4 Have error checking on input

The emphasis should be on preventing errors, versus handling errors. Instead of asking "what do I do if this error happens", ask the question "what can I do to keep this error from happening." For example, give the user "menu" choices whenever possible. Choices requiring a single digit or letter answer save errors due to misspellings, variations in capitalization, typing, etc.

If the user is asked to supply a directory name where a file can be placed, ensure the directory exists and is searchable and writable before trying to write a file there.

If the user requests copies of software source or executables, find out if the user has sufficient space before copying.

4.4.1.5 Have useful error messages

It is important to remember that error messages relevant to the software developer are rarely relevant to the user. Error messages should be in the user's terminology and reference any files, environment variables, or documentation that might help correct the error.

4.4.1.6 Avoid the use of "jargon"

Sometimes software developers frequently use terms the course of program development, that they no longer recognize the phrase as "jargon". For example, replace statements like "Please choose a filler" with statements like "Please choose a message generator".

4.4.1.7 Provide easy set up for environment and paths

There is a danger involved in setting up tool environmental variables in .cshrc. If the environmental variable is set to a rarely changed path, there is no problem. But if the variable is likely to take on different values, a problem could arise.

Example: RLF's Graphical Browser (GB) is run on the library identified by the environmental variable RLF_LIBRARIES. While running the GB, there are a variety of available actions. Some of these actions involve the invocation of programs run in separate shells. The .cshrc file is read at the beginning of execution by each shell. If the value of RLF_LIBRARIES is set in the .cshrc file, the value of RLF_LIBRARIES can change in the middle of execution. The user is unaware of the switch. Actions depending on the original value of RLF_LIBRARIES no longer work and the user has no idea what caused the breakdown. An alternative approach is to provide a script invoking the GB. In the script, the environmental variables needed by the program are placed together where they are easily identified and easily edited to accommodate local paths.

4.4.1.8 Avoid presenting irrelevant/illegal options

Only include relevant options in any listing of choices presented to the user. This may include only listing files with read/write access when appropriate.

Example: The system composition tool does a tree-walk of the semantic network to provide "choices" from which the user selects components. If a concept exists in the network, but the component at that concept does not currently exist in the library, do not offer the user that "irrelevant" choice. The constraint list, described above, can be used for this purpose.

4.4.1.9 Avoid distracting messages

Avoid distracting messages and flashed messages that can't be read. Allow the user to decide the pace of the tool. If the tool displays a message, let the user decide, through input, when execution should continue. Perform simple housekeeping chores on debugging statements before the code is released for general use.

4.4.1.10 Make it easy to stop and resume

Make it easy for the user to stop where he/she is and resume at the same place later. Users of a tool frequently do not know the answers to some of the questions. They may not know whether the X-Window System they wish to use is version 4 or 5. The user should not be obligated to repeat the entire selection process to register this choice. At appropriate points, choices should be saved in an incomplete state until the user is ready to resume the selection process.

4.4.1.11 Allow attachment of free text notes

Free text notes are useful for recording where you are, what you are going to do next, and rationales for why you did something. This concept has been included in several software architecture representations and allows users to record why a certain design decision was made.

4.4.2 Tool Coding Guidelines

Good coding practices should always be followed whether the product being developed is for onetime use or for reuse - because you never know what you may want to reuse later. The coding guidelines specified in Chapter 3 and Appendix D of this handbook, along with the general guidelines provided here can be a starting place for code standards.

4.4.2.1 Publish a style guide

If the development is not driven by a contract (that requires specific coding formats and standards), it is a good idea to publish a style guide which may include references tc accepted coding practices or styles specific to the project, computer language, or tool. The style guide can include formats for comments, error messages, color schemes, user interfaces, menus, etc.

4.4.2.2 Follow accepted coding standards

Use standard, or de facto standard, languages (e.g., Ada, C, C++). Refer to Appendix D of this handbook for guidelines in writing reusable code.

4.4.2.3 Confine domain-specific code

Confine domain-specific code to single identifiable files. Separate generic and instantiation specific code into different files. This separation should be transparent to the user. Changes to the instantiation specific files should not require recompilation of the code.

Example: As mentioned before, the system composition tool does a tree-walk of the modeled network to present the user choices. The tree-walk uses a generic algorithm, i.e., the tree-walk can begin at any node in the network; it performs the same no matter where it is begun. The tree-walk ends when all the nodes in the network beneath the starting node have been traversed and all of the choices involved with the components at those nodes have been made. The chosen component(s), on the other hand, often require additional specific information unattainable through the generic tree-walk. To acquire this information before a system is demonstrated or retrieved, the tool dynamically executes code specific to the component in question. The file in which the code resides matches the name of the node at which the component resides. This separation of generic and specific code helps in program maintenance. Like the constraint list, changes made to these files do not require recompilation of the tool.

4.4.2.4 Take advantage of tool building tools

Many compiler tools (e.g., aflex, ayacc, and expert system shells like CLIPS) make the job of building tools much easier than duplicating the functionality from scratch. Graphic User Interface (GUI) generators (e.g., Bluestone's uim/x and Builder Xcessory) can assist in cutting development time. Even developing front end tools for databases can decrease development time and increase productivity.

4.4.2.5 Use SCM Tools for version control

Use software configuration management (SCM) tools for version control. This is important when development is accomplished by more than one programmer. Most computer companies today require development using these tools so that the tool components can become part of the local or project library (an initial effort towards a reuse library).

4.4.3 Evaluation

Before tool testing can begin, the following design and code goals should be evaluated.:

Goal: Ensure an adequate tool has been designed and developed.

Questions:

- Does the tool perform in a clear and consistent manner?
- Does the user understand the purpose of the tool?
- Is the tool "user-friendly"?
- Does the tool provide instant feedback when needed?
- Is the tool free of annoying features?
- Is the tool easily to customize?
- Is the tool easily maintainable?
- Is the tool reusable?

4.5 Tool Testing

Invariably, applications undergo a formal process to verify that they fall within the specification guidelines by which it was created. Further, the application must also conform to the set of requirements that brought about the need for its development. The process of validation (producing the right product) and verification (producing the product right) faces applications in all domains, including reuse tools. This section discusses issues relevant to how much and how long testing should be undertaken. Further, focus is on how the availability of tool features or services are important in reuse processes and how the tool's interaction with other tools can affect its adoption into that process.

This section assumes that formal or government testing has not been specified. For information about testing relating to government testing (i.e., formal testing) refer to Section 3.6.

4.5.1 Adequate vs. Complete Testing

Typically a tongue-in-cheek response to the question "When is testing done?", usually boils down to either when resources for the software development are exhausted (we're out of money) or the product's delivery date has arrived (we're behind schedule) [WEYUKER89]. Unfortunately this actually has been the case for an untold number of projects. Projects ending in this manner rarely have high customer satisfaction or high confidence in the end product. This is due, for the most part, to insufficient planning for the amount of testing required to verify and validate the application.

It is clearly difficult to judge, up front, what amount of testing may be required to completely validate an application. [KEMMERER89] differentiates between completely validated and adequately validated, by inferring that the latter is more plausible than the former. He goes on to state that trying to answer when a system has been adequately validated and can be put into production cannot be answered in a absolute fashion. Highlighting the difference between adequately validating a random fortune (typical Unix utility) generated when a user logs out compared to adequately validating a real-time embedded system such as the Strategic Defense System emphasizes this point. This introduces the first critical factor necessary to determine 'adequately tested', i.e., the criticality of the tool being developed.

The next critical factor in determining the appropriate level of testing for a tool can hinge upon the targeted customer base. The level of testing applied to a tool developed for local or internal use will be significantly different than the resources applied to one for the commercial market. Further, this factor should be weighed based upon its heterogeneity of use. If the tool is targeted to many different heterogeneous environments, the amount of testing differs from that same tool targeted for only one system.

The last category one should consider is the complexity of the tool. This is meant to include the type of tool being developed (e.g., requirements analysis and design and code generator), its intended interaction or interoperability with other tools, standards which must be adhered to for the tool to be accepted (e.g., POSIX), and even the tools used in the process of developing the new tool (e.g., Ada compilers and static analyzers).

For large or complex systems, testing can consume about half of the overall development costs. Careful planning is essential to get the most of the development dollars. The critical factors identified above (criticality, customer and complexity) should be used to guide the development of a test plan. Additionally, the value applied to each of the critical factors and their associated weights vary for each type of tool and the domain for which it is intended. The test plan itself is concerned with describing the testing process rather than the actual tests themselves and should contain (but is not limited to) the following [SOMMERVILLE]:

- A description of the major phases of the testing process. Traditionally these phases are defined as unit, module, subsystem, integration and acceptance testing. Although it is recognized that these are typically defined in traditional waterfall methodologies, other software development methodologies (such as rapid prototyping and spiral development models) can adopt these phases.

- A description of how testing traceability to requirements is to be demonstrated. The customer is most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.

- An overall testing schedule and resource allocation for this test schedule.

- A description of the relationship (or interdependencies) between the test plan and other project plans. Simply stated, for the execution of a test in the test plan, describe what must be comp.eted for the test to be properly executed.

- A description of the way in which tests are to be recorded. This must be an auditable process to ensure the tests are being executed correctly and the results are properly being recorded for comparative analysis.

Further, the test plan sho' . ·· ··tain assumptions made about the testing process and the rationale used in determin.. dt· quantitative or qualitative values assigned to each critical factor. Emphasizing the critical .actors assist in the justification for measuring the adequacy of the amount of testing allocated for a particular tool.

4.5.2 Testing Techniques

The technique or methodology used to test a tool further impacts the amount of testing necessary. There are a variety of static and dynamic testing techniques. Generally, static techniques are applied to the design and coded implementation of the tool. Therefore detailed knowledge of the implementation of a tool can be very valuable in identifying appropriate test cases by which the tool implementor can play an important role. Static techniques employ program inspections (walkthroughs), mathematical program verifications (e.g., Anna, a verification tool for Ada) and program analyzers (e.g., lint).

Dynamic techniques exercise the tool, or features of the tool, in a simulated environment closely matching the developer's expectation of the operational environment or the process in which the tool is to automate. This involves running the tool, observing inputs and outputs, and looking for unexpected behaviors. This implies the dynamic aspects of the tool cannot be fully and rigorously tested until all the tool's components are complete and integrated to form the cohesive tool.

Whether static and/or dynamic techniques are applied in testing, specific trade-off regarding costs and time must be considered. White-box or path-testing typically requires Pn tests where P is the number of paths in a program and n is the number of decision points. Black-box testing, on the other hand, requires testing the total possible number of input values and combinations thereof (a combinatorial problem none the less) [DeMILLO89].

The black-box approach is taken by most tool integrators and end users of reuse tools. This shadows the model reuse engineers take to evaluating other types of components for reuse. Tool integrators evaluate based on the advertised features of the tool and their availability in the context in which they are intended to be used. Therefore for a tool developer and tester, black-box testing should be the minimalistic approach taken to testing. However, for the tool developer, quality assurance of the developed reuse tool usually requires a more in-depth technique to testing the tool. In most cases, the tool integrator is not privileged to the tool's design and source code; hence, more exotic tests are impractical.

4.5.3 Tool Functionality

Determining what to test is as important as the depth and breadth of the testing. The arduous task of determining the functionality of a tool to test can be mediated by decomposing the tool's advertised functionality against various dimensions. By examining a particular "slice" of the functionality and combinations of functions, clues can be identified for developing testing paradigms.

The first dimension considers the context in which the tool is intended to operate. This dimension identifies with what the tool may come into contact. Further, this highlights implicit and explicit requirements put on the tool. Explicit requirements are driven from the advertised features of the tool, whereas implicit requirement are derived from the interactions the tool has in an operational environment. Characteristics of this dimension are summarized below:

- platform (portability, compatibility, etc.),
- framework (interchange standards, graphical user interfaces, dialogue, etc.),
- tool interoperation (data formats, storage mechanisms, carrier mechanisms, etc.),
- human operator (usability, understandability, etc.),
- processes supported (course-grained, fine-grained processes, etc.).

The second dimension considers how the tool itself is architected. Specifically, this places constraints on how the tool can be expected to perform in an operational environment. Typically, the tool's architecture satisfies its explicit requirements (since these requirements are usually identified during requirements analysis). How the tool behaves in an operational environment, therefore its acceptability, may hinge on its conformance to implicit requirements. This dimension identifies various tool architectures [WALLNAU92b]. Virtually any tool is comprised of one or more traits from the following tool architectures:

- Data management:
  - **Data dictionary tools** are much like deriver tools, but provide data management and manipulation facilities to maintain the computation model of the transformation used during derivation. Tools such as IDE's Software Through Pictures and VAX/VMS's Ada Compilation System (ACS) are examples.
  - **Database tools** are like data dictionary tools in that they provide data management services. However, database tools also provide data management and manipulation services over the source files used in the transformation. Example of tools in this class are ReQuire and Rational's environment for Ada.

  - **Deriver tools** perform computational transformations to map from one data format to another such as Ada and C compilers, code generators and reengineering tools.
  - **Filter tools** such as Unix tools like grep and sort which are ambivalent to data source or destination.
- Operating system process-level:
  - **Client/Server tools** have the characteristics of both parent/child and persistent tools. The server is typically persistent while the client is transient. However, the client processes are not of the same process context as that of the server. MIT X11 Windows and Template Software's SNAP are good examples of client/server tools.
  - **Parent/Child tools** are similar to persistent tools except the parent is persistent and spawns children (i.e., transient tools). FrameMaker fits this model of tools, as do testing tools such as XCapBak from Software Test Works.
  - **Persistent tools**, unlike transient tools, remain active after completing their tasks, e.g., the standard Hewlett-Packard SoftBench environment's text editor).
  - **Transient tools** perform discrete functions with or without the service of other tools. Unix utilities and tools are mostly transient (e.g., make, cc, and ci).
- User Model:
  - **Multiple User** (multi-user license and floating license).
  - **Single User** (single user license).

Figure 4–2 illustrates tool functionality. Orthogonal to the first two perpendicular dimensions, is the third which is specific to the tool's advertised functionality. The functions, or features, may come from the tool being tested directly, or may be derived from the class of tools in which the tool belongs. Various intersections of the three highlight the available features in different contexts given specific tool architecture characteristics.

To derive a plausible test scenario, Ts, from this three dimensional matrix, select a tool context, Cx, tool functionality, Fy, and tool architecture, Az. Ts(Cx, Fy, Az) is a valid test scenario if Fy is intended to operate in Cx and is supported by Az. For example, a process idiom incorporating rapid prototyping would be hindered by a tool whose data dictionary architecture could not update or refresh its dictionary at a sufficient rate. Therefore Ts fails on Az.

One common testing fallacy is the testing of individual features in isolation. It is possible to individually test features to only find that when legally used in combination, they fail. A scenario depicting combinational feature testing is shown in the lower left corner of Figure 4-2. In this example a combination of features, Fy and Fy' are tested in the context of Cx which is supported by two architectural features of the tool, Az and Az'. A good example of this is commonly seen in compilers. Most compilers have an optimization feature and a language debugging feature. When each feature is tested in isolation, the tests meet acceptance criteria. However, when both features are used together, either the optimization goals are penalized or the debugging mechanism fails. Therefore Ts fails on Fy and Fy'.

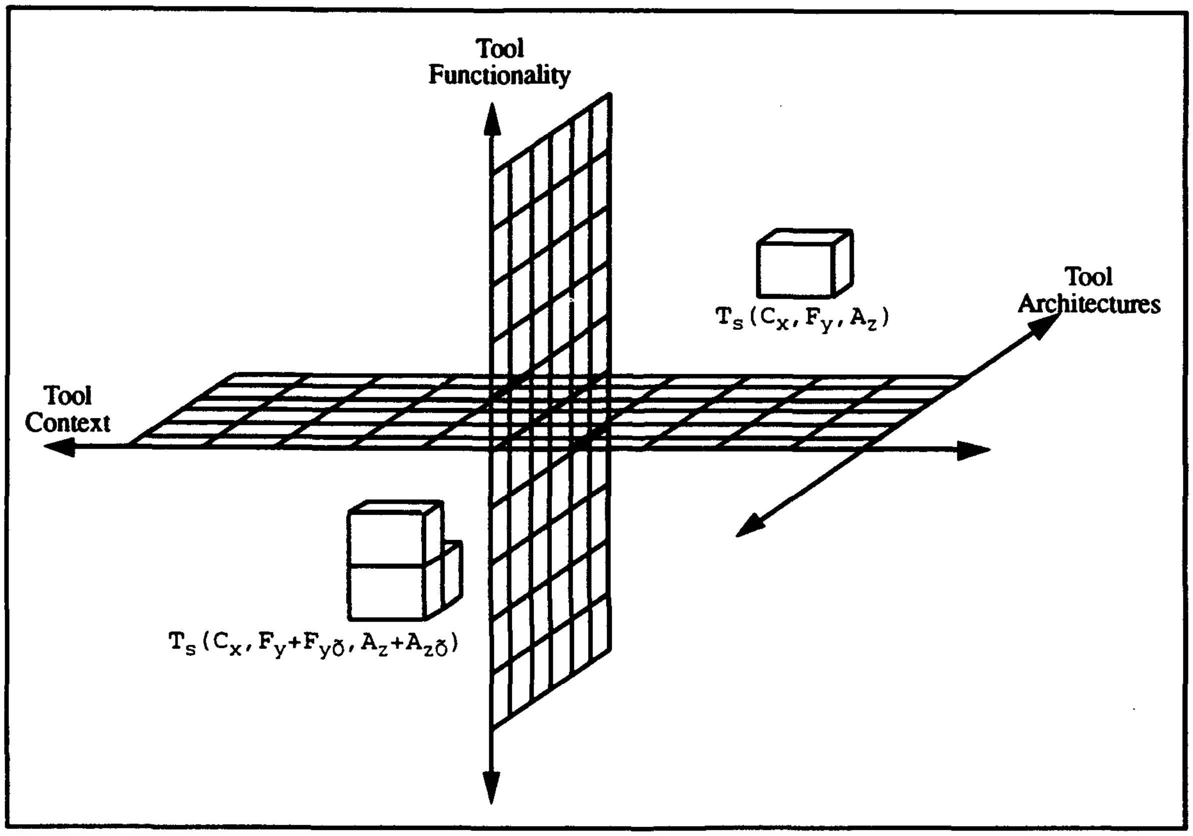

Figure 4-2 Tool Functionality

4.5.4 Tool Adoption

Applying a scenario-based approach to tool testing can further identify common areas that may hinder tool adoption. Primarily, this focuses on the tool's ability to interoperate with other tools and the tool's appropriateness for the process in which it is targeted.

4.5.4.1 Tool Interoperation

The capability for a tool to interoperate, or integrate, with other reuse tools used in a reuse process will be considered by tool integrators. Section 4.3.1 provides a sample of the plausible domain metrics for reuse tools. From the perspective of the tool integrators, this questionnaire provides insight on how the reuse tool will be judged in the context of the tool's ability to interoperate with existing tools. Test scenarios should be selected to address those metrics.

Tool interoperation is not only concerned with integration with other tools but also with the "human" tool. Additional domain metrics not included in section 4.3.1 may be needed to deal with specific human factors to be considered in the testing process. These factors could directly address the specific usability and understandability of the tool itself. This directly correlates to installation and user manuals as well as service or hotline support. Training courses also fall into this category. Textual or graphical user interface should be considered as well.

It is plausible that test scenarios include peer reviews of the manuals and services delivered to the end-user. Further, a one-on-one interaction with potential tool users provides valuable feedback on how the tool might actually be perceived by the end-user.

4.5.4.2 Tools and Processes

Testing the tool for interoperability with other tools, as well as the end-user, exercises the mechanics of the tool. To determine whether a tool is appropriate for a process, the need exists to address the conceptual abstractions the tool asserts about a given process and the granularity to which the tool is targeted. Specifically, is the tool focused on one aspect of the process, as in domain-modeling, or is the tool targeting an entire iteration of a process, like the system composition tool supporting rapid prototyping? It is also rational that the tool is targeting life-cycle component management, which is an integral part of any reuse process.

A fundamental problem arising when discussing the tool integration into a process is detecting the presence of abstraction and decomposition errors. [HOWDEN89] relates decomposition and abstraction errors to components of an application. This also applies to decompositions and abstractions made by tools about a particular process.

Decomposition errors occur when a tool developer makes a false assumption about the properties of a process. Abstraction errors occur when there is a mismatch (not between the mental abstractions, as in decomposition errors, but between some feature of the process and the abstraction used by the tool developer in their implementation of that feature).

To illustrate the last point, consider the following. A component management tool asserts that the concept of a component is mapped to the individual source file and entity level. The process the tool is targeting conceptualizes a component as being those elements in combination as the component. For example, in Sybase, each individual delivered part of the product (executable, data files, and configuration files) in itself a component, or is the product in its entirety the component? This fictional component management tool has made a decomposition error with respect to the process in which it intends to support. On the other hand, a generative tool, such as a code generator, may be developed to support reuse processes in a manner such that the tool can be highly reused; when in fact the product of the tool itself is what is intended to be reused. Although the high-level product representation can be reused, the code generator's somewhat cryptic output is too specific and cannot be reused. In this example the abstraction about the tool's usage in the process did not match the process's intended usage of the tool.

Another problem closely related to decomposition and abstraction issues is the granularity at which the tool is to be integrated into a process. [WALLFEIL91] provides a comprehensive discussion of tool integration issues from the perspective of the tool integrator rather than the tool developer. This unique perspective gives the tool developer insight to how the tool may be perceived by the tool integrator. Tool integrators identify that some tool integration strategies making sense under one process may not be reasonable under alternative process models. "Deciding which tool services to integrate, and how to integrate them, is dependent upon the organization's software processes being automated or supported. Conversely, tools themselves may support or impose process models having an impact on the organization's processes." [WALLFEIL91]

With respect to the tool developer, determining how to test a tool in the context of a reuse process will be greatly influenced by how the tool supports that process. Tools targeted for relatively more mature processes should tend to support and automate that process, whereas tools targeted for less mature processes may tend to impose their own model of that process (i.e., does the tool drive the process or does the process drive the tool?).

Consider the process of generating a system from the composition of components in a reuse library. A tool such as a system composition tool is designed to automate that process. If the process of composing a system takes a matter of weeks, the process would not be well supported by a tool in which the user is tied to the workstation and is required to complete the composition session due to the transient nature of the tool itself. Rather, the tool would better support the process if it followed a persistent, database architecture. With that, the composer can readily and incrementally save the work and return to it at a later date.

Clearly, the way tools are built directly correlates to the availability of services in a particular context. Devising test scenarios around the advertised tool functionality sensitive to the context in which the tool will be used, highlights flaws either in the tool's architecture (hindering its use) or identifies combinations of features inadvertently conflicting with one another. Further addressing the criticality of the tool in its targeted domain, the intended customer base and tool's complexity provides the rationale for determining the extent of test coverage as well as the techniques necessary to obtain sufficient, and therefore adequate, testing.

4.5.5 Evaluation

Before tool testing can be considered complete the following goal must be evaluated for success:

Goal: Verify and validate the advertised features of the tool are available within the context in which the tool is intended to be used.

Questions:

- Has the functionality of the tool been tested enough based on targeted users and their environments?

- Has the interoperability of the tool, with other tools in the reuse process, been tested?
- Has the tool's user interface been adequately tested?
- Does the tool cause measurable improvements in the reuse process?
- Does the test plan reflect the priorities assigned to the requirements?

4.6 Tool Maintenance

Tool maintenance, like component maintenance, depends largely on the policies and procedures specified by the reuse library. It is recommended that all tools developed, including those developed for internal use, are included in a library that will foster reuse. Once a tool is in a library, its maintenance is similar to that of component maintenance defined in Chapter 3.

APPENDIX A - REFERENCES

For a complete list of CARDS documentation email to hotline@cards.com or call the hotline at (800) 828-8161.

NOTE: References listed with an asterisk (*) are not referenced in the body of this document but are included as supplemental references.

[ABELOW93]	Abelow, Daniel, "Wake Up! You've Entered the Transition Zone". *Computer Language*, March 1993, Volume 10, Number 3, pp. 40-48.
[AF92]	*Air Force Software Reuse Implementation Plan*, Air Force Software Management Division, HQ USAF/SCXS, 24 Aug 92.
[AFOTEC89]	*Software Maintainability - Evaluation Guide*, AFOTEC Pamphlet 800-2, Volume 3, Department of the Air Force, HQ Air Force Operation Test & Evaluation Center, Kirtland Air Force Base, NM 87117, October 1989.
[AGGARWAL93]	Aggarwal, K., A Weighted Operational Profile, *Software Engineering Notes*, Jan. 1993.
[ARMOUR93]	Armour, J., Watts Humphry, *Software Product Liability*, Software Engineering Institute, CMU/SEI-93-TR-13, ESC-TR-93-190, Aug 93.
[ARMY92]	*The Army Strategic Software Reuse Plan*, 31 Aug 92.
[ARMY93a]	*Army Software Reuse Implementation Plan*, 7 Oct 93.
[ARMY93b]	*Communications-Electronics Command (CECOM) Advanced Planning Briefing for Industry*, 20 Oct 93.
[ASSET92]	Poore, J.H., Theresa Pepin, *Criteria and Implementation Procedures for Evaluation of Reusable Software Engineering Assets*, ASSET, The National Software Technology Repository, March 1992.
[ATKINSON91]	Atkinson, K. *Object-Oriented Reuse, Concurrency and Distribution: An Ada Based Approach*, Addison Wesley 1991.
[BAILIN92]	Bailin, Sidney C., "KAPTUR: Knowledge Acquisition for Preservation of Tradeoffs and Underlying Rationales." *CTA Tech News*, CTA Incorporated Technical Magazine, Special Issue, December 1, 1992.

[BAILIN93]	Bailin, Sidney C., "Domain Analysis with KAPTUR", *Tutorials of TRI-Ada '93*, Volume I, September 18-23, 1993. Association for Computing Machinery, Inc., New York, New York.
[BIGGER89]	Biggerstaff, T.J., A.J. Perlis, ed. *Software Reusability, Volume 1, Concepts and Models*, ACM Press, New York, New York, 1989.
[BLAN92`	Blankenship, A.B. and George Edward Breen, State of the Art Marketing Research, 92.
[BOEING92]	The Boeing Company. *Draft Reuse Maturity Model: Reuse Strategy Model Prototype, Software Technology for Adaptable, Reliable Systems (STARS)*, Document # D613-55159, 30 Nov. 1992.
[BOOCH91]	Booch, *Object-Oriented Design with Applications*, Benjamin Cummings 1991.
[BRACKEN92]	Bracken, Mike, *KAPTUR Release 1.0 User's Guide (Draft)*, April 29, 1992. Prepared for NASA Goddard Space Flight Center, Data Systems Technology Division, Applications Development Branch, Greenbelt MD, by CTA Incorporated.
[BRAUN92]	Braun, *DSSAs: Approaches to Specifying and Using Architectures*, STARS 92, Dec. 1992.
[BROWN93a]	Brown, A. "Control integration through message-passing in a software development environment", *Software Engineering Journal*, May 1993.
[BROWN93b]	Brown, Carney, Feiler, Oberndont, Zelkowitz "A Project Support Environment Reference Model", *TriAda 93*.
[BROWN94]	Brown, Carney, Morris, Smith, Zarrella, *Understanding CASE Tool Integration*, Oxford University Press (to appear in 1994).
[BUCK93]	Buck, "Knowledge for Sale! The advent of industry-specific class libraries.", *IEEE Expert*, Oct 1993.
[BUSCHMANN93]	Buschmann, "Rational architectures for object-oriented software systems", *Journal of Object-Oriented Programming*, Sept. 1993.
[CALDIERA91]	Caldiera, Basili, "Identifying and Qualifying Reusable Software Components" *IEEE Computer*, Feb. 1991.

[CARDSa] *Acquisition Handbook*, Central Archive for Reusable Defense Software (CARDS), STARS-VC-B011/001/00, 25 March 1994.

[CARDSb] *Direction Level Handbook*, Central Archive for Reusable Defense Software (CARDS), STARS-VC-B012/001/00, 25 March 1994.

[CARDSc] *Engineer's Handbook*, Central Archive for Reusable Defense Software (CARDS), STARS-VC-B002/002/00, 28 February 1994.

[CARDSd] *Technical Concepts Document*, Central Archive for Reusable Defense Software (CARDS), STARS-VC-B009/001/00, 29 January 1994.

[CARDSe] *Library Development Handbook*, Central Archive for Reusable Defense Software (CARDS), STARS-VC-B005/001/00, 29 Oct 93.

[CARDSf] *Library Operation Policies and Procedures for the Central Archive for Reusable Defense Software, Volume I*, STARS-VC-B004/002/00, 28 February 1994.September 1993.

[CARDSg] *Library Operation Policies and Procedures for the Central Archive for Reusable Defense Software, Volume II*, STARS-VC-B004/002/01, 28 February 1994.

[CARDSh] *Model Contracts/Agreements*, STARS-VC-B014/001/00, 25 Mar 94.

[CARDSi] *Command Center Library Model Document*, Release 3.2.1, Central Archive for Reusable Defense Software (CARDS), STARS-VC-B015/003/ 00, 12 Nov 92.

[CARDSj] *Training Plan*, Central Archive for Reusable Defense Software (CARDS), STARS-VC-B003/001/00, 29 January 1994.

[CARDSk] *Franchise Plan*, Central Archive for Reusable Defense Software (CARDS), STARS-VC-B010/001/00, 28 Feb 94.

[CARDSl] *CARDS Software Architecture Seminar Report*, STARS-VC-B008/001/00, 29 January 1994.

[CARDSm]* *CARDS Domain Model Document*, STARS-AC-04110/001/00, Nov. 1992.

[CARDSn]	*Command Center Library Model Document*, Release 3.2.1, Central Archive for Reusable Defense Software (CARDS), STARS-VC-B015/003/ 00, 12 Nov 92.
[CARDSo]	*Command Center Library Model Document*, Release 3.2.1, Central Archive for Reusable Defense Software (CARDS), STARS-VC-B015/003/ 00, 12 Nov 92.
[CARDSp]	*Component Tool Developer's Handbook*, Central Archive for Reusable Defense Software (CARDS), STARS-AC-04114/001/00, 15 March 1993.
[CARDSq]	*Command Center Library User's Guide*, Release 3.2.1, Central Archive for Reusable Defense Software (CARDS), STARS-VC-B006/003/00, 12 Novemeber 1993.
[CARDSr]	*Command Center Version Description Description*, Release 3.3, Central Archive for Reusable Defense Software (CARDS), STARS-VC-B007/004/00, 25 February 1994.
[CCDH87]	*Command Center Design Handbook*, Defense Communications Agency, May 1987.
[COHEN90]	Cohen, S., *Ada Support for Software Reuse*, CMU/SEI-90-SR-16.
[COHEN92]	Cohen, Stanley, Peterson, Krut, *Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain*, CMU/SEI-91-TR-28, June 1992.
[COX90]	Cox, B "Planning the Software Industrial Revolution", *IEEE Software*, Nov. 1990.
[CSRO92]	*DoD Domain Definition Report*, DoD Center for Software Reuse Operations, Document # 1222-04-210/28, 15 Jul. 1992.
[DeMILLO89]	DeMillo, Richard A., "Test Adequacy and Program Mutation" In *11th International Conference on Software Engineering*, IEEE Computer Society Press, Washington D.C., 1989.
[DISA92a]	*DoD Domain Definition Report*, DoD Center for Software Reuse Operations, The Defense Information Systems Agency/Corporate Information Management (DISA/CIM), May 1992.

[DISA92b] Defense Information Systems Agency/Corporate Information Management (DISA/CIM) *Technical Reference Model*, Version 1.2, 15 May 1992.

[DISA93a] *Domain Analysis and Design Process*, Version 1, Software Reuse Program, DISA/CIM, Document No. 1222-04-210/30.1, CDRL 30.1, 30 Mar 93.

[DISA93b] DISA, *Software Reuse Program Software Reuse Metrics Plan*, Version 4.1, 4 August 1993.

[DIXON92] Dixon, R., *Winning with CASE Managing Modern Software Development*, McGraw-Hill 1992.

[DoD88] *DoD Defense System Software Development*, DOD-STD-2167A, 29 Feb 1988.

[DoD92] *DoD Software Reuse Vision and Strategy Document*, #1222-04-210/40, 15 July 1992.

[DOLAN94]* Dolan, Kevin, "PROTOTYPES: Tools That Can Be Used and Misused", *CrossTalk*, January 1994.

[DUMASS88] Dumas, J., *Designing User Interfaces for Software*. Prentice Hall 1988.

[ELLEMTEL92] M. Henricson and E. Nyquist, *Programming in C++, Rules and Recommendation*. Ellemtel Telecommunication Systems Laboratories, April 1992.

[EVB93] Goldfedder, Brian, "An Internal Approach to Testing Horizontally Reusable Components", EVB Software Engineering, Inc., published in the *Proceedings of the Fifth Annual Software Technology Conference*, Salt Lake City, UT.

[FEDPUBS] *Selling Software to the Government*, Federal Publications, Inc.

[FEILER93] Feiler, *Reengineering: An Engineering Problem.* CMU/SEI-93-SR-5 July 1993.

[GARLAN93a] Garlan, Shaw, "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering*, vol.1 1993.

[GARLAN93b] David Garlan and Curtis Scott, "Adding Implicit Invocation to Traditional Programming Languages" *Proceedings*

	of The 15th International Conference on Software Engineering, May 17-21, 1993 Baltimore, MD, pp. 447-455.
[GAUTIER90]	Gautier and Wallis, *Software Reuse with Ada*, IEE Peter Peregrinus, 1990.
[HISSAM93]	Hissam, Scott, "CARDS Trip Report on UNAS Training." Not published but available through CARDS Program Office.
[HOLIBAUGH92]	Holibaugh, R., "Domain Analysis Products". *STARS Newsletter*, Sept. 1992 vol.III no. 2.
[HOLLING92]	Hollingsworth, "Software Component Design-for-Reuse: A Language Independent Discipline Applied to Ada," Dissertation, Ohio State University, 1992.
[HOWDEN89]	Howden, W. E., "Error-based Validation Completeness", Appeared in the *11th International Conference on Software Engineering*, IEEE Computer Society Press, Washington D.C., 1989.
[IEEE92]	"Tools Assessment" issue of *IEEE Software*, May 1992.
[IEEE STD 90]	*IEEE Std 610.12, IEEE Standard Glossary of Software Engineering Terminology*, Dec. 1990.
[IEEE STD 1175]	IEEE STD 1175, "A Trial-Use Standard Reference Model for Computing System Tool Interconnections", 1991.
[KANG90]	K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.
[KANG92]	Kang, K. C., S. Cohen, R. Holibaugh, J. Perry, A. S. Peterson, *A Reuse-Based Software Development Methodology*, Software Engineering Institute, CMU/SEI-29–SR-4, Jan 92.
[KAPTUR]	KAPTUR *Release 1.0 User's Guide*, Draft CTA, Incorporated, prepared for NASA Goddard Space Flight Center under Contract Number NAS5-30680, April 1992.
[KARAT92]	Clare-Marie Karat, Robert Campbell, and Tarra Fiegel, "Comparison of Empirical Testing and Walkthrough Methods in User Interface Evaluation". *CHI '92 Conference Proceedings*, ACM Conference on Human

Factors in Computing Systems, A CM Press, New York, 1992, pp. 397-404.

[KATZ93] Katz, Susan, Christopher Dabrowski, and Margaret Law, *Glossary of Software Reuse Terms - Draft*, Department of Commerce, October 28, 1993.

[KEMMERER89] Kemmerer, Richard A., "Compi [illegible]y Validated Software."Appeared in the *11th International Conference on Software Engineering*, IEEE Computer Society Press, Washington D.C., 1989.

[KOLTUN91] Koltun, P., and A. Hudson,, "A Reuse Maturity Model." *Proceedings of the 4th Annual Workshop on Software Reuse*, Nov. 1991.

[KRUEGER92]* Krueger, Charles, "Software Reuse", *ACM Computing Surveys*, Vol 24, Number 2, June 1992.

[LEVINE92]* Levine, Trudy, "Reusable Software Components", *Ada Letters*, 1993.

[LONG93] Long, Morris, An overview of PCTE: A basis for a Portable Common Tool Environment, CMU/SEI-93–TR-1, ESC-TR-93–175.

[METTALA92] Mettala, E., Graham, M. "The Domain Specific Architecture Program." In *Proceedings of the DARPA Software Technology Conference*, April 1992.

[MUSA87] Musa, Iannino, Okumoto, *Software Reliability Measurement, Prediction, Application.* McGraw-Hill 1987.

[NATOa] *Development of Reusable Software Components*, NATO Communications and Information Systems Agency, Volume 1.

[NATOb] *NATO Standard for Management of Reusable Software Component Library*, NATO Communications and Information Systems Agency, Volume 2 of 3.

[NATOc] *Software Reuse Procedures*, NATO Communications and Information Systems Agency, Volume 3 of 3.

[NAVY93] *Reuse Implementation Plan*, Preliminary Draft, LSIS 930020, Department of Navy, Naval Information Systems Management Center, Space and Naval Warfare Systems Command, 29 Jan 93.

[NIELSEN93a]	Nielsen, Jakob, "Iterative User-Interface Design", *Computer*, November, 1993, pp. 32-41.
[NIELSEN93b]	Nielsen, Jakob, "Is Usability Engineering Really Worth It?" *IEEE Software*, November, 1993, pp. 90-92.
[NIERSTRASZ92]	Nierstrasz, Gibbs, Tsichritzis, "Component Oriented Software Development", *Communications of the ACM*, Vol. 35, No. 9 Sept. 1992.
[NIST93]	National Institute of Standards and Technology, *Glossary of Software Reuse Terms (version 1.2)* - Draft, October 28, 1993.
[NRL91]	*NRL Guidelines for Ada Portability and Reusability*, Unisys Feb. 1991.
[OMG92]	OMG, "Object Management Architecture Guide" Sept. 1992 and "The Common Object Request Broker: Architecture and Specification" 1992.
[PAULK93a]	M.C. Paulk, B. Curtis, M.B. Chrissis, and C. V. Weber, *Capability Maturity Model for Software, Version 1.1*. Software Engineering Institute, CMU/ SEI-93-TR-24, February 1993.
[PAULK93b]	M.C. Paulk, C. V. Weber, S. Garcia, M.B. Chrissis, and M. Bush, *Key Practices of the Capability Maturity Model, Version 1.1*. Software Engineering Institute, CMU/SEI-93-TR-25, February 1993.
[PERMAR93]	Permar, Dave, *Legal Issues and Reuse Workbook*, Reuse Education and Training Workshop, 25-27 Oct 93.
[PERRY92]	Perry, Wolf, "Foundations for the Study of Software Architecture", *ACM SIGSOFT SEN*, Oct. 1992.
[PETERSON93]	Peterson, Stanley, *Mapping a Domain Model and Architecture to a Generic Design*. CMU/SEI-TR draft.
[POORE93]	J. H. Poore, Carmen J. Trammell, Charles Snyder, Laurence Van Dolsen, *SDP-2000: A Guide to Project Implementation of Megaprogramming*, Software Engineering Technology for IBM Corporation, CDRL 04050-001, Mar 93.
[PRIETO-DIAZ87]	Prieto-Diaz, Ruben, "Domain Analysis For Reusability", in *Proceedings of COMPSAC 87*, Tokyo, Japan, 23-29 October, 1987.

[PRIETO-DIAZ90]	Prieto-Diaz, Ruben, "Domain Analysis: An Introduction", *ACM SIGSOFT Software Engineering Notes*, vol. 15, no. 2, April 1990.
[RAPID91]	"Qualifying Software Components: Reusability Metrics", in *Software Reuse and Re-Engineering Conference for the National Institute for Software Quality and Productivity*, 30 April, 1993.
[REUSE93a]	*Proceedings of the Second Annual West Virginia Reused Education Training Workshop*, 25-27 Oct 93. Hosted by: AdaNet, ASSET, CARDS, DISA Software Reuse Program, and West Virginia University.
[REUSE93b]	*Proceedings: Software Reuse Legal Issues Workshop*, (22-24 Mar 93), STARS-AC-04117/001/00, 30 Apr 93.
[RMM92]	"Towards a Reuse Maturity Model", *Proceedings of the 5th Annual Workshop on Software Reuse*, Oct. 1993.
[SAGE90]	Sage, A., and J. Palmer, *Software Systems Engineering*, John Wiley & Sons, New York, 1990.
[SAIC93]	Workmann, Dr. David, "An Approach to Implementing Reusability Guidelines for Ada", SAIC, ASSET Operation, published in the *Proceedings of the Fifth Annual Software Technology Conference*, Salt Lake City, UT.
[SAUNDERS93]	T. Saunders, B. M. Horowitz, M. L. Mleziva, *A New Process for Acquiring Software Architecture*, The MITRE Corporation, Bedford, MA, 1993.
[SEI87]	Software Engineering Institute, *A Guide to the Classification and Assessment of Software Engineering Tools*, CMU/SEI-87-TR-1.
[SEI92]	K.C. Lang, S.Cohen, R.Holibaugh, J.Perry, A.S. Peterson, *A Reuse-Based Software Development Methodology*, Software Engineering Institute, CMU/SEI-29-TR-4, Jan. 1993.
[SOMMERVILLE90]	Sommerville, Ian, *Software Engineering, 3rd ed.* Addison-Wesley Publishing Co., Reading, MA, 1990.
[SPC92a]	*Ada Quality and Style: Guidelines for Professional Programmers*, Software Productivity Consortium, SPC-91061-CMC, Version 2.01.01, Dec. 1992.

	Software Productivity Consortium, Inc., SPC Building, 2214 Rock Hill Road, Herndon, Virginia, 22070.
[SPC92b]	*Reuse Adoption Handbook*, Software Productivity Consortium, Virginia Center of Excellence, SPC-92051-CMC, Version 01.00.03, Nov. 1992.
[SSC90]	"Recommended C Style and Coding Standards", Distributed by Specialized Systems Consultants, Inc. (SSC), P.O. Box 55549, Seattle, WA 98155 (206) 527-3385. Rev 6.0, 25 Jun 1990.
[STARS90]	*STARS Reusability Guidelines*, IBM, April 1990.
[STARS92]	*STARS Reuse Concepts, Volume 1, Conceptual Framework for Reuse Processes (CFRP), Version 2*, STARS-UC-05159/001/00, 13 Nov. 1992.
[STARS93]	*STARS Organization Domain Modeling (ODM) Volume 1 - Conceptual Foundations, Process and Workproduct Descriptions, Version 0.5 - DRAFT*, STARS-UC-05156/024/00, July 31, 1993.
[STRAKER92]	Straker, David, *C-Style: Standards and Guidelines*. Prentise Hall, New York, 1992.
[STSC92]	Software Technology Support Center, *Requirements Analysis and Design Tools Report*" April 1992.
[TRACZ91]*	Tracz, Will, "A conceptual model for megaprogramming", *Software Engineering Notes*, ACM Press, Vol. 16, Number 3, July 1991.
[TRACZ93]	Tracz, "Megaprogramming and Domain Engineering", *ICSE 15 Tutorial Notes*, May 1993.
[VITALETTI92]	W. G. Vitaletti, R. Chhut, *Domain Analysis Guidelines*, SofTech (DoD Software Reuse Initiative, The Defense Information Systems Agency/ Corporate Information Management (DISA/CIM), May 92.
[WALLFEIL91]	Wallnau, Kurt C., Feiler, Peter H., *Tool Integration and Environment Architectures.* Technical Report, CMU/SEI-91-TR-11, Software Engineering Institute, Carnegie Mellon University, March 1992.
[WALLNAU92a]	K. Wallnau, *CARDS: A Blueprint and Environment for Domain-Specific Software Reuse*, Central Archive for

	Reusable Defense Software (CARDS), STARS 92 Conference, 8 Dec. 1992.
[WALLNAU92b]	Wallnau, Kurt C., *Issues and Techniques of CASE Integration with Configuration Management.* Technical Report CMU/SEI-92-TR-5, Software Engineering Institute, Carnegie Mellon University, March 1992.
[WARTIK92]	S. Wartik, R. Prieto-Diaz, "Criteria for Comparing Reuse-Oriented Domain Analysis Approaches", *The International Journal of Software Engineering & Knowledge Engineering*, Sep. 1992.
[WEYUKER89]	Weyuker, Elaine J., "In Defense of Coverage Criteria", *11th International Conference on Software Engineering*, IEEE Computer Society Press, Washington D.C., 1989.
[WITHEY93]	Withey, *Implementing Model Based Software Engineering in your Organization: An Approach to Domain Engineering*, CMU/SEI-TR draft, 1993.

APPENDIX B - ACRONYMS/ABBREVIATIONS

B.1 Introduction

This appendix includes acronyms and abbreviations used in this handbook, it is not intended to be a complete list of reuse acronyms and abbreviations.

B.2 Acronyms and Abbreviations

4GL	Fourth Generation Language
AdaKNET	Ada Knowledge NETwork; semantic network representation scheme of the RLF which is written in Ada
AdaTAU	Ada Think Ask Update; rule based knowledge representation scheme in the RLF
ANSI	American National Standards Institute
ARPA	Advanced Research Projects Agency
ASL	Application Specific Language (a high level language for generating source code)
ASSET	Asset Source for Software Engineering Technology
CAIS	Computer Aided Instruction Software
CARDS	Central Archive for Reusable Defense Software
CASE	Computer Aided Software Engineering
CBD	Commerce Business Daily
CCL	Command Center Library
CIM	Center for Information Management
CLIPS	C Language Integrated Production System (a NASA product)
CMM	Capability Maturity Model
COBRA	Common Object Request Broker Architecture
COTS	Commercial Off The Shelf
DBMS	Database Management System

DFARS	Defense Federal Acquisition Regulation Supplement
DISA	Defense Information Systems Agency
DoD	Department of Defense
DOD-STD	Department of Defense Standard
DSRS	Defense Software Repository System
DSSA	Domain Specific Software Architecture
FAR	Federal Acquisition Regulation
FIPS	Federal Information Processing Standards
FODA	Feature-Oriented Domain Analysis
GDL	Graphical Design Language
GKS	Graphical Kernel System (graphic interface system)
GOSIP	Government Open Systems Interconnection Protocol
GOTS	Government Off The Shelf
GPLR	Government Purpose License Rights
IEEE	Institute of Electrical and Electronics Engineers
KAPTUR	Knowledge Acquisition for Preservation of Trade-offs and Underlying Rationales
MBSE	Model Based Software Engineering
NASA	National Aeronautics and Space Administration
NIST	National Institute of Standards and Technologies
OO	Object-oriented
OMG	Object Management Group
OS	Operating System
OSF	Open Software Foundation
PDL	Program Design Language
POSIX	Portable Operating System Interface for computer environments

PRISM	Portable, Reusable, Integrated Software Modules; a STARS Program
RFP	Request for Proposal
RIG	Reuse Library Interoperability Group
RLF	Reuse Library Framework, STARS supplied CARDS library mechanism
SBIR	Small Business Innovation Research
SEI	Software Engineering Institute
SGML	Standard Generalized Mark-up Language (text encoding standard)
SOW	Statement of Work
SPO	System Program Office
SQL	Structured Query Language
STARS	Software Technology for Adaptable, Reliable Systems

APPENDIX C - GLOSSARY

C.1 Introduction

This Glossary contains terminology used in this handbook. It is not intended to be a complete glossary of reuse terms. For an additional glossary of software reuse terms refer to [NIST93].

C.2 Terms

Ad-hoc Reuse	Reuse is practiced ad-hoc when there are no defined methods for performing reuse.
Adaptability	A measure of the ease with which a component can be altered to fit differing user images and system constraints.
Architecture-Centric Reuse	Reuse is architecture centric when the component development process and the application development processes are based on a generic architecture. The goal of an architecture-driven process is to achieve black-box reuse.
Binding	Language specific interface to the services defined in a standard (a wrapper for components written in a different language).
Black Box	Electronic equipment/software that functions and is packaged as a unit and whose internal mechanism is hidden from the user.
Black-box Reuse	Black-box reuse is achieved when application engineers can compose systems by plugging together different reusable components based on an application's requirements.
Certification	See Component Certification.
Command Center	A facility from which a commander and his representatives direct operations and control forces. It is organized to gather, process, analyze, display and disseminate planning and operational data and to perform other related tasks.
Component	A set of reusable resources related by virtue of being the inputs to various stages of the software life cycle, including requirements, design, code, test cases, documentation, etc. Components are the fundamental elements in a reusable software library.

Component-Based Library	Component-based libraries are similar to book libraries. They can be thought of as software warehouses. The central focus of a component-based library is the component.
Component Certification	The process of determining that a component being considered for inclusion in a library meets the requirements of the library and passes all testing procedures. Evaluation takes place against a common set of criteria (reusability, portability, etc.).
Component Qualification	The process of determining a potential component is appropriate to the library and meets all quality requirements. Evaluation takes place against domain criteria.
Contract Liability	A contract basis of liability is derived from agreements which might be oral or written (preferably written). Contract liability results when one party or the other breaches a term (provision) of the contract [REUSE93b].
Copyright	Copyright protection applies to the expression of an idea. Copyright protection remains in effect for the author's life plus fifty years and applies to copying a work, and not an independent creation. Copyright law allows copying of software for backup and archival purposes [REUSE93a].
Disclaimer	A written statement purporting to negate or limit potential liability. Disclaimers are used as a method of controlling potential liability by reducing the number of situations for which an entity can be held to have breached a duty or contract.
Domain	An area of activity or knowledge containing applications which share a set of common capabilities and data.
Domain Analysis	The process of identifying, collecting, organizing, analyzing, and representing the relevant information in a domain based on the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within the domain.
Domain Engineering	An encompassing process which includes domain analysis and the subsequent construction of components, methods, tools, and supporting documentation that address the

	problems of system/subsystem development through the application of the knowledge in the domain model and software architecture.
Domain Model	A definition of the functions, objects, data, and relationships in a domain, consisting of a concise representation of the commonalities and differences of the problems of the domain and their solutions.
Domain-Specific Reuse	Reusing components in a specific domain to build an instance of an application in that domain.
Domain Specific Software Architecture	A software architecture that has been generalized based on actual and projected commonalities of systems in a domain. May also be referred to as a generic architecture.
Generic	An Ada package or subprogram allowing parameters of unspecified type to promote code reuse. Generics are instantiated with actual parameters at compile time.
Generic Architecture	High-level paradigms and constraints characterizing the commonality and variances of the interactions and relationships between the various components in a system. May also be referred to as a domain specific software architecture.
GKS (graphics)	Graphical Kernel System (GKS) is a set of basic functions for computer graphics programming usable by many graphics-producing applications. This standard allows graphics application programs to be easily transported between installations; aids graphics applications programmers in understanding and using graphics methods; and guides device manufacturers on useful graphics capabilities.
GOSIP (network services)	Government Open Systems Interconnection Profile (GOSIP) defines a common set of data communication protocols that enable systems developed by different vendors to interoperate and the users of different applications on those systems to exchange information.
Hold Harmless Provision	Written assumption of liability by one party whereby he/she agrees to protect another party from anticipated claim(s).

Horizontal Domain	The knowledge and concepts that pertain to a particular functionality of a set of software components that can be utilized across more than one application domain.
Indemnification	Written promise of a party to insure; to secure against loss or damage that may occur in the future.
Intellectual Property	An intangible output of a rational thought process which has some intellectual or informational value (e.g., computer software, cinema, sound recordings) [REUSE93a]. Intellectual property can be protected by patents, copyrights, and trade secrets.
Government Purpose License Rights	The right to use, duplicate, or disclose technical data (applies to computer software under the Small Business Innovation Research (SBIR) program), in whole or in part, and in any manner, for government purposes only, and to permit others to do so for government purposes only [REUSE93a].
Knowledge Blueprint	A flexible plan to transition knowledge to the community.
Large Scale Reuse	Large scale reuse is the reapplication of high-level components (e.g., requirements, architectures, designs).
Library-Assisted Reuse	Reuse is library-assisted when there exists a library to support the application domain. There may be more than one library and they may be interconnected.
Limited Rights	The right to use, duplicate, or disclose in whole or in part or for the government, with the express limitation the computer software documentation shall not be used to prepare the same or similar computer software and shall not be released outside the government, except when needed for emergency repairs.
Megaprogramming	Megaprogramming is achieved when systems and subsystems can be viewed as black-boxes that meet certain requirements. These systems can be reused in building other systems without the developer requiring detailed knowledge of the system's internal structures.
Model-Based Library	Model-based libraries are organized around the principle that what matters in a repository is the context in which reusable software components are used and the relationships among components. The focus of a model-based library is the model (requirements, architectures,

	design decisions and rationales) and the software which implements these models.
Motif (window manager)	OSF/Motif was designed by members of the Open Software Foundation (OSF) as a standard graphical interface that would work identically on a wide variety of platforms, from high-end PCs to mini- and super-computers. Motif mainly consists of a toolkit for programmers.
Negligence	Conduct falling below the standard established by law to protect against unreasonable risk of harm; harmful defects that could have been detected and corrected through reasonable quality control practices [ARMOUR93].
Non-Disclosure Provision	Written agreement reflecting mutual assent between parties as to the confidential nature of information disclosed upon the stated condition that its release will be prohibited beyond a described person or persons, and limited to the use(s) stated.
Opportunistic Reuse	Reuse is practiced opportunistically when it is up to software developers to identify when reuse is possible, locate reusable components, and integrate them.
Ownership	Holding title to tangible property (e.g., owner of an automobile under state issued certificate of title) [PERMAR93].
Patent	Patents protect ideas and give the inventor the exclusive right to prevent others from making, using or selling the invention for seventeen years after the patent is issued [PERMAR93]. (Note: patent protection for software is not only new, but is controversial).
POSIX (UNIX OS)	Standard Portable Operating System Interface for computer environments (POSIX) defines a standard operating interface and environment based on the UNIX Operating System documentation, to provide for the portability of applications software at the source-code level, between computer systems from multiple vendors.
PostScript (printer)	The PostScript language, developed by Adobe Systems Incorporated, is designed to be a modern standard for electronic printing. Its primary application is to describe the appearance of text, graphical shapes, and sampled

	images on printed pages. The description is high-level and device-independent.
Process-Driven Reuse	Software reuse is process-driven when it is an integral and transparent part of both the software engineering process and the broader acquisition process.
Qualification	See Component Qualification.
Restricted Rights	Applies to computer software, and includes, as a minimum, the right to: use the computer software with the computer for which it was acquired, including use at any governmental installation to which the computer may be transferred by the government; use computer software with a backup computer if the computer for which it was acquired is imperative; copy computer software for safekeeping or backup purposes; and modify computer software or combine it with other software.
Reusable Components	Domain model, software architecture, product design, and implementation components (source code, test plans, procedures and results, and system/software and process documentation).
Reuse Library	A library specifically designed, built, and maintained to house reusable components.
SGML (text encoding)	Standard Generalized Markup Language (SGML) specifies a language for document representation. SGML can be used for publishing in its broadest definition, ranging from single-medium conventional publishing to multimedia database publishing.
Small Scale Reuse	Small scale reuse is the reapplication of code: subroutines, object libraries, or Ada packages.
Software Architecture	A specification for the assemblage of components.
SQL (DBMS)	Structured Query Language (SQL) is the relational database language used to create, store, modify, retrieve, and manage information in a relational database. The SQL standard has been adopted by the American National Standards Institute (ANSI) and the International Standards Organization (ISO).
Statutory Liability	A statutory basis for liability requires neither a contract nor the existence of any duty under tort law, but is covered under statute or laws (e.g., copyright law).

Strict Liability	A part of tort law that covers damage caused by or threatened by unreasonably dangerous products; the product contains one or more unreasonably dangerous defects [ARMOUR93].
Systematic Reuse	Reuse is practiced systematically when there exist defined procedures for leveraging future software projects. Efforts are devoted up-front to creating a suitable process.
Tort Liability	A duty to provide a certain "standard of care" which is expected from another party. This is derived from the common law of tort.
Trade Secret	Any formula, process, design or intellectual property interest which is protected by secrecy [REUSE93a].
Unlimited Rights	The right to use, duplicate, release, or disclose, technical data or computer software in whole or in part, in any manner and for any purpose whatsoever, and to have or permit others to do so.
Vertical Domain	The knowledge and concepts that pertain to a particular application domain.
Warranty	Assure customers that the products will perform as stated; can limit the supplier's liability in the event of nonperformance [ARMOUR93].
White Box	Electronic equipment/software that functions and is packaged as a unit and whose internal mechanism is known to the user.
Wrapper	1. A component which allows passing of data between components. 2. A script which allows RLF access to components.
X11 (window system)	The X Window System was developed by Project Athena, a Massachusetts Institute of Technology (MIT) development team working in association with Digital Equipment Corporation (DEC) and International Business Machines (IBM). It is a graphical network environment which is hardware and vendor independent.

APPENDIX D - DEVELOPING REUSABLE COMPONENT

D.1 Introduction

This appendix discusses an overview of guidelines and considerations for use when developing reusable code - code intended to be used again on a different project from that on which it was developed. The issue of standards and guidelines for developing reusable code is a current issue in the software research community, and continuing research will likely modify and improve upon the guidelines discussed below. This appendix is not intended to be an in-depth survey of the current state-of-the-art, but rather a brief overview of some of the major issues involved.

Reuse guidelines can be partitioned into two main divisions:

- Language Independent Guidelines (section D.2) - guidelines independent of programming language.

- Language Specific Guidelines (section D.3) - guidelines dependent upon the computer language with which the software was developed.

D.2 Language Independent Guidelines

The software should be developed to achieve low coupling or dependence with other software components of the system and high cohesion within the reusable software (i.e., the software should be self-contained and its components should perform simple single functions). Low coupling refers to the degree of interdependence between separate software modules. High cohesion refers to a software module performing a single, well-defined function.

The source code should be extensively commented with text describing requirements the software solves, the design of how those requirements are solved, and the details of any interface to external software, systems or variables. Documentation providing details in greater depth than embedded comments allow should be produced and should be available.

The following applies to all software, even if source code is not available (e.g., COTS):

- Minimize and document known system dependencies. The creator of reusable software cannot always predict the situation in which the software will be reused. Frequently, software developed for Unix based workstations is reused in embedded system real-time development. Or, perhaps, is used on machines where the word size and internal representation of the data elements causes unpredictable behavior. For example, not all machines use an 8-bit byte or use two's complement arithmetic. Dependencies on these features can be subtle and hard to detect.

- The software's original developer should document the hardware and software environment under which the software was originally developed and should describe the hardware platform, tools, and operating system in detail.

- Reusability concerns are closely related to long term maintainability issues. The Air Force publishes a language independent guide to software maintainability issues addressing many of the concerns required for the development of reusable software [AFOTEC89]. This guide discusses maintainability issues by describing their rationale and gives examples in several computer languages.

D.3 Language Specific Guidelines

Each computer language has its own set of recommended practices and style of coding. Adherence to a published standard makes the software easily understood by an unfamiliar user, and thus increases its reusability. For many languages the published language reference manuals provide information on standard use of the language features and can serve as a starting point for standardizing coding rules.

Adherence to standards facilitates system maintenance and modification. The following guidelines apply to all software systems even, if the source code is not available (e.g., COTS):

- The Software Productivity Consortium (SPC) style guide is the DoD recommended standard for the computing language Ada [SPC92a]. This guide describes formatting and indentation as well as recommendations for the use of specific language features. It has been adopted as the official style guide for all DoD programs. Science Applications International Corporation (SAIC) has developed a program, the Ada Software Reuse Assessment Tool (ASRAT) [SAIC93], which performs syntactic and semantic analysis for Ada source units with respect to a subset of the guidelines provided in [SPC92a].

- Ellemtel Telecommunication Systems Laboratories has developed a style guide [ELLEMTEL92] for the computing language C++. This guide includes both style rules and detailed descriptions of their recommended use of C++ features. It provides detailed examples of recommended guidelines with sample code provided to illustrate the rationale. These examples illustrate subtle errors avoided by use of the style guidelines.

- [SSC90] is a 40 page style guide and coding standard for use with the C programming language. This guide contains references to 12 additional programming guides for C [STRAKER92].

APPENDIX E - SAMPLE REUSE TOOL DESCRIPTIONS

E.1 Introduction

This Appendix discusses a sampling of reuse tools. By no means is this to be considered a complete list of reuse tools. The following reuse tools are discussed in this appendix:

- Reuse Library Framework (RLF) (section E.2)
- CARDS System Composition Tool (section E.3)
- CARDS Qualification Tool (section E.4)
- KAPTUR (section E.5)
- UNAS/SALE (section E.6)
- InQuisiX™ (section E.7)
- Ada-ASSURED™ (section E.8)

E.2 Reuse Library Framework (RLF)

The knowledge representation scheme used in RLF is described in some detail in this section since it is the basis for other reuse tools, including the CARDS System Composition Tool:

- **CARDS Modeling Concepts**. Though a domain analysis may be conducted to a large extent without regard to the final form of its products, the knowledge acquired cannot be considered a domain model until it is harnessed in some formalism. RLF is the mechanism used to implement the CARDS libraries. RLF integrates a knowledge representation scheme, rule-based inferencing and a graphical browser. The description of the nature of the model requires an overview of the encoding mechanisms. The discussion below is a conceptual level description of how the model is encoded. Simplified examples are given, not necessarily following the actual syntax of the tool.

- **AdaKNET**. RLF has a knowledge representation scheme, called AdaKNET, facilitating the classification of library components. AdaKNET is a semantic network, but can be thought of as a graph in which the nodes represent general categories or specific objects and the edges represent relationships between the nodes. Nodes, representing general classes of knowledge, are called concepts. There are two basic types of relationships, specialization ("is-a") and aggregation ("has-a"), described below. Two kinds of hierarchies are built with the concepts and relationships:

- **Specialization**. There is exactly one specialization hierarchy in any given "model". Model is a word which potentially has a very specific meaning when used in a very narrow context, but is often used somewhat loosely. It will be used in a very general way in this handbook. The specialization hierarchy is a way of defining the concepts. It can be thought of as a glossary and can be compared to the need to declare variables in a traditional computer programming language. The top level concept is usually given a very generic name, such as thing or entity, and all other concepts are defined to be specializations of it. This hierarchy is made by asserting the "is-a" unidirectional relationship holds between two concepts. The assertion is that "less general concept specializes more general concept" or less general concept "is-a" more general concept.

  For instance, animal specializes thing, mammal specializes animal, and cow specializes mammal. The modeler can build a set of concepts representing the parts of the domain needing to be referenced. The specialization hierarchy provides the vocabulary for an AdaKNET model. Because every concept in the model is defined in this hierarchy as being a specialization of some other concept, there is a way of understanding the context of any concept encountered in the model.

- **Aggregation**. Each of the concepts in the specialization hierarchy may have relationships of the "has-a" nature with any other concept in the model, including itself. These relationships may express attributes, characteristics, features (as the term is used in FODA), functional capabilities, requirements or metrics - any relationship existing between two concepts which is not a specialization relationship. Since any concept may have an aggregation hierarchy under it, and it is not required that all the units of aggregation structure tie together, the model usually contains many separate pieces of aggregation hierarchy. While every concept must appear in the specialization hierarchy, it is not necessary for every concept to be involved in an aggregation relationship.

  The aggregation hierarchy is built by beginning with the concept representing the thing being modeled, such as command center, and listing all of its "has-a" relationships. These relationships show substructure, characteristics or other sorts of relationships and are often called roles. Each of these relationships has a name, type and range. The type is any concept. The range is an ordered pair, zero to infinity, or some narrower specification, including a converged range such as 2 to 2. This value represents how many copies of that relationship may/must exist simultaneously.

- **The Model**. The above two types of hierarchy (specialization and aggregation) are completely intertwined. In most cases what the modeler and the end user think of as "the model" is actually a subtree (really a subgraph, but it is generally thought of as a tree) rooted at the concept representing the thing to be modeled and all the aggregation hierarchy under it.

Each concept automatically inherits the aggregation relationships of its ancestors in the specialization hierarchy. AdaKNET supports multiple inheritance, so a concept having more than one parent inherits the aggregation relationships of each. The range, and the possible values of the type, may be narrowed on subsequent levels of the hierarchy (by the concepts inheriting them) to support the logical structure of the aggregation hierarchy (known as: role restriction).

A full model includes much more than the structural organization expressed in AdaKNET; with the inferencers, discussed below, being the most important other part. However, the focus of reuse tools being discussed is on the structural part of the model. This is because the structure is the aspect most usefully described. Discussing why the structure was organized as it was, communicates something about the modeler's understanding of the command center domain. In contrast, the inferencers are aids for accomplishing tasks.

- **Inferencing**. Two inference engines have been used with AdaKNET. AdaTAU was written in conjunction with AdaKNET and is a part of the RLF. It is tightly integrated with the Graphical Browser. CLIPS (C Language Integrated Production System) was developed for NASA and is available for a very moderate fee, or free for use on Air Force or NASA projects. These two inference engines provide similar basic capabilities, but CLIPS is more computationally powerful and also has the capability of querying the AdaKNET model structure. Both inference engines permit the modeler to write inferencers, units of code expressed as rules about the model. These units, sometimes called rule-bases, are associated with specific concepts in the AdaKNET model.

E.3 CARDS System Composition Tool

The CARDS System Composition Tool is an evolving tool. A description of the tool follows:

- **System Composition.** One of the main purposes of a domain model is to capture information to help a software engineer build a system. In CARDS, the capability to interactively build a system from reusable components by capitalizing on information in the domain model is called system composition. A tool performing system composition is invocable directly from the RLF version of the CARDS library models. The following discussion describes the basic operation of this tool:

  System composition is reuse in action. The user is asked questions about what system functionalities they need. These questions are based on the components currently available in the library and the relationships among the components (represented in the model). Each answer may trigger the process of filling in related implementation details. The approach used in the System Composition Tool is a form of knowledge-based software engineering. Rules for composing a system are written in CLIPS and used for inferring what questions to ask from information contained in AdaKNET and the answers provided by the user. System composition does the following basic process:

- Input:
  - CARDS Library Model.
  - User knowledge of desired system.
- Output:
  - Appropriate source and/or executable code extracted from the reuse library for the desired system.
  - Demonstrations of selected systems/subsystems.
- Processing Sequence:
  - User clicks on "Perform Action" at a buildable concept node (i.e., system/subsystem). Note: Some nodes cannot be composed into a system because they cannot function as stand-alone entities.
  - CLIPS queries AdaKNET to retrieve roles to be filled. For example, a message generator subsystem has roles for user interface and message formats. CLIPS determines from the model what versions of components are available to implement the user interface and message formats.
  - CLIPS asks the user domain dependent questions relevant to the particular system being composed; thus allowing all necessary roles to be filled.
  - If the preconditions for building the chosen node are met (i.e., all questions are answered), CLIPS then harvests the system by collecting all the role filler information and generating calls to shell scripts which put the source/executable code together in user specified directories.

E.4 CARDS Qualification Tool

The CARDS automated qualification tool is designed to determine whether a prospective component is suitable for inclusion into the library model. The underlying network used for "qualification" is the modeled RLF network which the tool uses to generate a component specific domain metric questionaire. The questionaire is interactively completed by the user with yes/no answers as to whether the prospective component has attributes indicated by the library model.

The qualification tool queries the RLF network for the node's attributes, attribute ranges, etc. After a question is completed, the user has the option of allowing corresponding code to be produced and added to the model script, thereby placing the component in the library. Any deficiencies discovered during the qualification process are noted in the generated code.

Automated qualification tool user interface is written in CLIPS. CLIPS contains only procedure code and does not utilize rule-based (AdaTAU or RBDL) information.

The qualification tool assumes the user wants to add the component at the node from which the "qualify component" action is invoked. Super-roles (aggregation or "has-a" relationships) of the invoked node are identified and the corresponding sub-roles (differentiated aggregation relationships) are retrieved and presented to the user in a questionaire.

Critical roles are required relationships having a range (1 .. n) in which the minimum number is at least one. A range (0 .. n) is an optional or non-critical relationship. Critical component features mapping directly to the critical roles in the library model are identified on the questionaire. A component then "qualifies" into the library model if it contains all of the critical component features.

The qualification tool uses the following logic:

1. Retrieve the roles for the node.

2. For each super-role (differentiated role), retrieve its sub-roles.

E.5 KAPTUR

The following is excerpted from the *Introduction of the KAPTUR (Knowledge Acquisition for Preservation of Trade-offs and Underlying Rationales) User Guide* Section 1.2, "What is KAPTUR?" [KAPTUR]:

> *"KAPTUR is a tool designed to be part of a reuse-based software development environment. A reuse-based software development environment is distinguished from a traditional software development by virtue of its explicit support for building new systems using components from previous systems as opposed to starting from scratch. Such an environment possesses a huge potential for building systems cheaper and faster. Generally, reuse is most effective when practised within a domain, a family of similar systems. Some examples of domains are:*
>
> - *POCC (Payload Operations Control Center Software),*
> - *CMS (Command Management System),*

- *Flight Dynamics,*
- *Networking,*
- *DBMS (Database Management System).*

KAPTUR can assist in reuse from the early stages of software development as opposed to waiting until the coding phase. For reuse to be most effective, it has to start from the early stages.

KAPTUR keeps representations for and knowledge about the components from past systems. It is this common representation and the knowledge from past systems that would assist you in comparing and contrasting components and figuring out which particular ones to reuse. Components themselves exist outside KAPTUR. But KAPTUR will tell you things like where to find them and who to approach. So, in a nutshell, first use KAPTUR to determine which components to reuse and where to get them, then get the components and give them a try.

When you browse through a database looking for things to reuse, we say that you are using KAPTUR in the demand mode.

You may be wondering about two things now. First, what form do the representations and knowledge mentioned above take, and second, how do they get in to KAPTUR. Let us first tell you about the form of these representations and knowledge and then about how they get in to KAPTUR.

We represent a system in KAPTUR as an architecture. An architecture tells you how the system was designed. We keep multiple views of an architecture. Each view tells you things about a different aspect of the system. Some of the views supported are Entity-Relation Diagram View, Classification Diagram View, and Assembly Diagram View. Diagrams for these views look similar to the ones you will generate while doing software development using object-oriented techniques. Of course, there are minor variations in syntax as well in semantics among many popular object-oriented diagramming methods. Again, just as you will create a hierarchy of diagrams while doing software development, we store our architectures in a hierarchical fashion.

The knowledge portion starts out with a list of special things about architectures. We use the term "features" for these special things. Thus a feature may represent a significant design or programmatic decision, or it may represent the use of a particular technology or method for a system whose architecture is kept in KAPTUR. A feature is listed as a brief phrase. Underneath this abbreviated level lie details which we refer to as explanations for features. An explanation for a sin-

gle feature contains the decision represented by the feature, what trade-offs were considered while making the decision, and the rational(s) for making the decision.

The architectures representing past systems in a given domain, and their features and explanations are put into KAPTUR by a person called a domain developer. A domain developer looks at the components (requirements, design, code, etc.) from these systems and populates KAPTUR's database for this domain. The domain developer also creates one or more generic architectures for this domain. The generic architectures represent common and recommended development practices for this domain. Now the domain developer can go back and add additional features to past system architectures. These features represent deviations from generic architectures. These features not only represent significant things about systems but also deviations from recommended approaches.

When a domain developer creates the representations and knowledge in KAPTUR, we say he is using KAPTUR in the supply mode. Only a domain developer is allowed to add information to KAPTUR databases. But any software developer can act as a supplier in his own database. Each user can create his personal database for any domain and experiment with creating a new architecture. This provides a bridge from selecting potential components for reuse (through browsing in KAPTUR) to actually acquiring and using them. A user may go back and forth between browsing and creating a new architecture in his database before making the final selection. Browsing can provide only a limited familiarity with architectures. It is only after we start to work with them that we begin to feel comfortable in reusing them. Thus a user database provides an interim work place."

E.6 UNAS/SALE

The following text is extracted from [HISSAM93]:

"UNAS (Universal Network Architecture Services) is a commercial product from TRW. UNAS is a process-based, asynchronous, message-driven, language framework for rapidly developing distributed (potentially heterogeneous) network applications.

As presented, UNAS is intended to fit into the megaprogramming software development paradigm as championed by the DoD STARS program. UNAS provides a standard applications programming interface to facilitate portability between heterogeneous platforms (they termed it as middleware). UNAS itself is based on common, open-systems standards such as POSIX, TCP/IP, and Ada.

Conceptually at its core, UNAS is simply (possibly an understatement) a distributed extension of the Ada programming language (with all the tools to build and manage a system built with UNAS). If you were to extend Ada's concept of tasking and rendezvousing (message-passing between tasks) to processes and message-passing you might begin to *get a basic understanding of UNAS's underlying concept. Since UNAS is a programmatic extension of Ada, you maintain all the features of the Ada language and essentially extend Ada strong typing, exception handling, and tasking to distributed processes across a (potentially heterogeneous) network of hosts.*

First a bit about UNAS's terminology. A network represents the processes (running on nodes) and the circuits which interconnect those process to form the distributed application. Therefore, a network is made up of one or more nodes, to which process are allocated. Each process is made up of one or more tasks which can receive zero or more messages via a socket. Sockets are defined as ports which can be defined as In, Out or InOut sockets. Sockets are tied to other sockets to form circuits. Messages pass over circuits (and therefore to other tasks or processes on other nodes in the network). Two other concepts are groups (to which processes belong) and timers. Processes usually belong to groups, but processes in a group do not necessarily have to execute on the same node. Timers are delayed writes of messages to a socket which is tied to itself. More details are in the course material.

UNAS is comprised of three product layers: Messaging Product, Architecting Product and CASE Tool Option Product.

The UNAS message product, provides the basic services for Inter-Task Communication (ITC) capability known as ITC services and automatic heterogeneous data translation. ITC supports four levels of inter-task communication:

- *Level 1 - process communicates with itself (timer sockets and state transitions);*
- *Level 2 - process to process communication within the same network node;*
- *Level 3 - process to process communication where each process resides on different, homogeneous, network nodes;*
- *Level 4 - process to process communication where each process resides on different, heterogeneous, network nodes.*

The messaging product provides tools and ITC services for heterogeneous data interoperability. Data structures written in Ada source are converted to Meta-message format via UNAS off-line message registration tools. The Meta-message format is the mechanism that permits data conversion between heterogeneous network nodes. Further, ITC services provide the Ada package "generic" to ensure Ada's strong type cohesion between the distributed processes which write and read passed messages.

Other services of ITC include error reporting and propagation, task creation, interactive network management, SNMP interface to network management, and message interjection and recording. Two significant clarifications are necessary regarding that last statement. First - the SNMP interface to network management specifically refers to registered SNMP variables and objects that can be accessed and manipulated via a SNMP client (such as

SunNet Manager) to monitor and control object within the UNAS network (remember - the UNAS network is that collection of nodes, process, tasks, and connections that make the distributed, hence networked, application). Second, message interjection and recording (MIR) is a facility, or capability, of UNAS to interject test messages into the application network and record network behavior for regression testing and performance monitoring.

The architecting product includes the messaging product and provides Generic Application Control (GAC). Again, in the simplest of terms, GAC is a higher level of abstraction of the services provided by ITC. Pragmatically, GAC removes the application developer from many of the "quirks" and details of the ITC layer as well as adding buffer I/O to network message passing, message queuing, logical separation of nodes, processes and sockets from their physical implementation, and built-in performance and utilization. Additionally, exception handling, error reporting and logging is greatly enhanced and abstracted in the GAC layer.

The GAC layer is the foundation, or stepping stone, to the CASE Tool Option Product. The CASE Tool, SALE (Software Architect's Lifecycle Environment), provides a high level architecture definition of the objects, relationships and attributes of design elements in a UNAS network application (hence a software architecture abstraction of the network application). SALE is comprised of three major components, the graphical editing front-end, the code-generating back-end, and UNAS's underlying GAC and ITC services.

The graphical front-end is a third party product called Virtual Software Factory Analyst Work Bench (vsf.awb). The vsf.awb provides a graphical user interface to the architectural elements in UNAS that make up an application network and the rules to put such an application network together. From the graphical representation of a UNAS architecture, SALE can check the structural semantics and consistency of the UNAS architecture and directly generate Ada source code which represents the graphical "source". The Ada sources generated contain the appropriate ITC and GAC Ada package instantiations, timing and performance delays to simulate expected "hand-crafted" code, and initialization and configuration files necessary to start the network application.

After code-generation, relatively minor changes need to be applied to the new Ada source code to actually get a "skeleton" application network up and running. Since SALE generates all the necessary code for the distributed portion of the application network, SALE advertises that it removes the necessary, but tedious, computer science practices of building distributed applications and the pitfalls of deadlocks and race conditions as well as representing and passing complex data structures across homogeneous (let alone heterogeneous) computing environments.

These early prototypes of the designed application can be used to experiment with various logical and physical architecture implementation to select an optimal implementation. The architecture selected then serves as the baseline implementation in which mission specific code is then added to the generated code to produce the fielded systems."

E.7 InQuisiX

InQuisiX™ is expected to be released in early 1994. This product has not been used by CARDS at this point but is included here as a sample reuse tool. The information contained in this section represents pre-release information and is therefore subject to change without notice.

InQuisiX is an adaptable classification and search engine that, when integrated with a software development environment, is an advanced software reuse library system. InQuisiX provides performance classification, cataloguing, searching, browsing, retrieval and synthesis capabilities assisting in reuse automation. Diverse kinds of reusable assets, including source and object code, program design language, graphics designs, documentation, unit development folders and tests, can be managed using InQuisiX. The various kinds of software components are organized by the user into a hierarchical class structure organizing the common and different attributes of component classes. Attribute definitions defined in higher level classes are inherited by lower level classes.

InQuisiX includes the concept of a classification scheme describing the information to be retained for each component being cataloged to organize, categorize, search, understand, evaluate, extract, adapt and integrate the component.

InQuisiX supports functions shown in Table E-1.

For more information about InQuisiX contact:

Software Productivity Solutions, Inc.
P.O. Box 361697
Melbourne, FL 32936
Phone: (407) 984-3370
FAX: (407) 728-3957

Table E-1 InQuisiX Features

Function	Key Features
Classification Schemes	• Faceted classifications (controlled vocabulary) • Keyword indexing (uncontrolled vocabulary) • Text indexing (information retrieval) • Characteristics-based attributes • Relationships between components
Cataloguing Facilities	• Forms-entry of component information with user-defined layouts • Bulk text and graphic attributes remain externally stored • User can integrate custom tools for automated derivation and cataloging of component attributes • Custom migration tools can move component information from existing libraries • Component information can be moved between libraries using the import/export facility
Search Mechanisms	• Iterative querying, associating new queries against component sets obtained from previous queries • Boolean query, used for precise searching • Forms-based query with user-defined layouts, used for faceted or template queries • Wildcard text matching, to handle possible term variants • Powerful browsing facilities, for viewing component attributes, related components, and text and graphic assets

Function	Key Features
Component Retrieval and Synthesis	• User kept set provides a shopping basket for collecting components of interest • Component attributes can reference representations structured for synthesis into system baselines (e.g., object code, graphic images, formatted document segments) • Users can integrate custom retrieval, adaptation and synthesis tools, for example to support graphic design synthesis
Open Architecture supports integration into user's development environment	• Component information can be stored and managed external to the reuse library system, in files, databases or configuration management systems in the user's environment • User tools or distributed reuse library systems can share information with the reuse library system via published file exchange formats for import/export • External tools can be invoked from within the reuse library system using a transparent tool invocation facility • Cooperating tools can manipulate component information via a process-to-process communications mechanism with the InQuisiX Server • The native storage of InQuisiX can be rehosted onto other management facilities using Classic-Ada with Persistence™

E.8 Ada-ASSURED

Ada-ASSURED™ is a multi-buffer, multi-window, language-sensitive editor developed by GrammaTech, Inc. and Loral Aerospace Corporation. It is a reuse tool in that it ensures syntactic correctness by enforcing compliance with accepted Ada programming style. Ada-ASSURED is designed to enforce *Ada Quality and Style Guidelines* developed by the Software Productivity Consortium (SPC) [SPC92a] and recommended by the Ada Joint Program Office (AJPO). Violations of SPC guidelines are detected incrementally as programs are developed and modified. Ada-ASSURED is also a programmable batch tool using a command language that allows controlled access to all editing and navigation commands in non-interactive mode.

For more information about Ada-ASSURED contact:

GrammaTech, Inc.
One Hopkins Place,
Ithaca, New York 14850
Phone: (607) 273-7340
FAX: (607) 273-8752
email: admin@grammatech.com

APPENDIX F - SOURCES OF REUSABLE COMPONENTS

F.1 Introduction

The following information is subject to change, without notification, and is not intended to be complete nor up-to-date.

F.2 Reusable Software Components

Column appeared in **Ada Letters** [LEVINE92]. Provided by Trudy Levine, Fairleigh Dickinson University Teaneck, NJ 07666 levine@sun490.fdu.edu levine@vax.fdu.edu.

Ada in Action is a set of 77 source files described in the book "Ada in Action with Practical Programming Examples", by D.W. Jones. The set includes:

- utilities expressing floating-point numbers in fixed or floating-point notation and convert free-form character input to a floating-point number, three portable user interfaces giving the application program complete cursor control, permit line editing and default responses, and support context-sensitive help,
- three file utility programs demonstrating file I/O and user interface techniques,
- dimensional data types catching subtle engineering application programming errors at compile time.

  These source files are supplied on two 5.25" IBM PC format floppy disks for $49.95, ISBN: 0–471-50747-4. CONTACT: John Wiley & Sons, Inc. P.O. Box 6792 Somerset, NJ 08875-9976.

Ada Language System/Navy (ALS/N) includes documentation and executable code for ALS/N Common Ada Baseline (CAB) available to contractors and agencies having contractual requirements to use these systems. **CONTACT**: Syscon Corp. 9841 Broken Land Parkway Columbia, Maryland 21046 Lisa Davis (410) 381-8300.

Ada Math Advantage consists of 106 frequently used components for signal processing, image processing, and linear algebra. Access is open, with full source code, test data, and installation verification supplied to licensees. Software is warranted and supported. **CONTACT**: Quantitative Technology Corp. (QTC) 9360 SW Gemini Drive Beaverton, OR 97005 Kenneth Cramer (503) 626-3081 TELEX 9102402827 FAX (503) 641-6012.

The AdaNET Repository is a component of the Repository Based Software Engineering (RBSE) Program, a research and development program designed to contribute to a significant advancement in the U.S. capability to affect timely transfers of software engineering technology among government, industry and academia organizations. AdaNET is an on-line repository of abstract data types, benchmark tools, CAIS tools, communication software, database management tools, graphics software, management tools, math components, metric analyzers, preliminary design tools, and text management tools. These assets, available on magnetic tape and diskettes, may also be downloaded via Internet and MountainNet direct dial-in (set up account through Ms Lacey below) with a VT100 compatible terminal. Test documentation is included with some components. Software is public domain and available "as is". Direct uploading or editing is not allowed. **CONTACT**: AdaNET Client Service MountainNet Inc., 2705 Cranberry Square Morgantown, WV 26505 Peggy Lacey (304) 594-9099, 1-800-444-1458 lacey@rbse.mountain.net.

ADASAGE is an applications development set of utilities designed to facilitate rapid, professional construction of Ada systems. Applications may vary from small to large multiprogramming systems utilizing special capabilities, including database storage and retrieval (interactive and imbedded SQL is provided), graphics (standard GKS is provided), communications (GOSIP), formatted windows, on-line helps, sorting, editing, etc. The ADASAGE application development system is available for Alsys, Meridian and Verdix compilers and runs under Unix and MS-DOS. ADASAGE is a non-proprietary government domain product not requiring a license for development or application sites. It may be obtained from (a group or handling fee may be required): ESTSC the primary distribution organization to obtain ADASAGE. A signed Software License form (requiring the name of the requestor, requesting organization and location site) is required to purchase the package. **CONTACT**: ADASAGE Users Group Idaho National Engineering Laboratory P.O. Box 1625 Idaho Falls, ID 83415; Warren O. Merrill (208) 526-0656 RAPID; Joanne Piper Arnette (703) 285-9007 Energy Science Technology and Software Center, P.O. Box 1020, Oak Ridge, TN 37831; Edwin M. Kidd, (615) 576-2606, (FTS) 626-2606, FAX: (615) 576-2865: verification (615) 576-2606; E-Mail: estsc@adonis.osti.gov.

AdaSoft Textual User Interface (TUI) provides fully integrated window, menu and forms managers, and is operational on DOS, various UNIX platforms and VAX/VMS; source code is available on tape or disk:

- AdaSoft Textual User Interface/Graphics (TUI/G) provides all of the facilities of the TUI in graphics mode; includes additional capabilities for drawing shapes, text, graphs, and PCX images in windows.

- Relational Database Management System (Ada manager) is currently on DOS.

- Ada Real Time PC Tool Kit, a collection of components accessing PC hardware is available on DOS. One-year maintenance is included in license; updates at optional cost.

  CONTACT: AdaSoft, Inc., 8750-9 Cherry Lane Laurel, Maryland 20707; Paul Maresca or Jerry Horsewood; (301) 725-7014 FAX (301) 725-0980.

ASR (Ada Software Repository) is a repository of Ada programs, software components, and educational material and has been established on the Defense Data Network (DDN), specifically to promote the exchange and reuse of Ada programs and tools. Software and documentation is in the public domain and provided "as is." Over 100 megabytes of Ada source code and information reside in the ASR. Besides the inclusion of Ada components, the following items have been added: PDL to DoD-STD-2167A translator, Educational information, DoD-STD-2167A document templates, Ada 9x, Kermit 3.01 communication program, latest newsletters, Ada style handbook, and pretty printer Ada tutor portable text formatter. **CONTACT**: on tape: The DECUS Program Library, 219 Boston Post Road BP02, Marlboro, MA 01752-1850; (508) 480-3418; on CD-ROM: ALDE Publishing, 6520 Edenvale Blvd., Suite 118; Eden Prairie, MN 55346; Bill Summins (612) 934-4239; on disk: Advanced Software Technology, Inc. (see below), 5 Patricia Lane Patchogue, NY 11772; Jeffrey Hickey (516) 758-6545.

Advanced Software Technology distributes STARS foundation products and the Ada Software Repository. STARS is a DoD research initiative to invent and demonstrate a new software engineering technology and acquisition practice for large systems in the Ada programming language. The products cover such software technology areas as: operating systems, DBMS, user interfaces, command languages, communications, graphics, text processing, run-time support, planning and optimization, and reusability assistance. Disks are available in high density (1.2 megabyte) format only and require a hard drive for uncompressing the files. All files are in the public domain and distributed "as is." **CONTACT**: Advanced Software Technology, Inc., 5 Patricia Lane, Patchogue, NY 11772, Jeffrey Hickey (516) 758-6545.

ASENTO (Ada Software Engineering Tools Project) consists of the tools: Ada Yacc (YACCA), which is an Ada version of the well-known UNIX tool yacc, and Adaface, an interface generator for Ada programs and packages. Both programs are written in Ada and produce Ada code. Programs have been made available on host kaarne.cs.tut.fi (130.230.3.2) and by anonymous ftp, directory pub/src/ASENTO. In case of problems **CONTACT**: Hannu-Matti Jrvinen, Tampere University of Technology, 33101 Tampere, Finland hmj@cs.tut.fi.

ASSET (Asset Source for Software Engineering Technology) was organized by ARPA's STARS Program. The responsible prime contractor is IBM, with day-to-day operation subcontracted to SAIC. ASSET's reuse library was established to serve as a national resource for the advancement of software reuse across DoD. ASSET's mission is to provide a distributed support system as a focus for software reuse within DoD and to help foster a software reuse industry within the U.S. Its goals are to create a focal point for software reuse information exchange, advance the technology of software reuse processes, provide a national emporium for reusable software products, and stimulate a national software reuse industry. ASSET's initial focus will be on Ada mission critical software components. ASSET operates the STARS Reuse Library, the STARS Bulletin Board, and provides Newsgroup services. The STARS Reuse Library consists of a repository and a bibliography. The repository currently contains three collections: the NRL STARS Foundation products from the Naval Research Laboratory, the STARS catalog products, a set of reusable software assets produced by the STARS prime contractors, and the Orlando software reuse library collection. The bibliography contains abstracts and other information needed to extract a desired reusable component from the library. Although the Newsgroup and some of the products have limited access, the STARS Bulletin Board, containing postings of the abstracts of the STARS products and events of interest to the software reuse community, is cleared for public distribution. These services can be reached via Internet (192.121.125.10) or by direct dial-in: **CONTACT**: ASSET telnet source.asset.com (304) 594–3635 (8 bits, no parity, 1 stop), enter STARSBBS at the login prompt; 2611 Cranberry Square, Morgantown, WV 26505; Larry Jacowitz, Director (304) 594-3954/1762 FAX: (304) 594-3951 info@source.asset.com.

The **Booch Components** consist of several dozen domain independent data structures and tools, each with multiple implementations so that a client can select the representation providing the most suitable time and space characteristics. The Booch components are written in Ada; a version in C++ in also available and a Smalltalk version is planned. The Ada version is currently in use at over 300 sites in the U.S., Europe, and the Pacific Rim. **CONTACT**: Rational, 2171 S. Parfet Court Lakewood, CO 80227; Grady Booch (303) 986-2405.

The **CAMP (Common Ada Missiles Packages)** Program is an on-going research and development program established in September 1984 at the McDonnell Douglas Corp. in response to an Air Force contract. It has produced over 500 tested Ada parts, as well as tools to support reuse. It includes CAMP parts, CAMP Armonics Benchmarks and CAMP Parts Engineering System. Top-level computer software components are grouped into data constants; data types; and equipment interface, navigation, Kalman filter, guidance and control, non-guidance, and general utilities. Media is ANSI standard labelled 9-track 1600 bpi tapes. CAMP products are not certified; source code is distributed and code can be tested via the drivers provided in the CAMP benchmarks. CAMP products are export controlled; only government activities and qualified contractors may receive CAMP. Source code may not be redistributed, but products using CAMP can distribute executable images. Tape is distributed in a VAX VMS format, but products are appropriate for any Ada environment. **CONTACT**: Data & Analysis Center for Software, P.O. Box 120 Utica, New York 13503; (315) 336-0937.

The **CARDS (Central Archive for Reusable Defense Software)** Program is applying existing state-of-the-art technology to provide an implementation framework for reuse libraries in domains of interest to the DoD, and applying this framework to the command center domain. The implementation of the CARDS Command Center Library (CCL) in Fairmont, WV, is based upon two technologies: Reuse Library Framework (RLF) and AFS. RLF is a knowledge-representation framework designed specifically for use as the basis of domain-specific libraries. AFS is a wide-area network file system. The CCL consists of reusable components at multiple levels of abstraction, including architectures, requirements, subsystems, and individual components (ranging in size from simple interfaces to large commercial products). Access to CARDS is restricted to government personnel or contractors. This may be opened up in the future. Most of the CARDS documents are "distribution A", i.e., can be distributed to the public. **CONTACT**: CARDS, 1401 Country Club Road, suite 201, Fairmont, WV 26554; (304) 367-0421, 1-800-828-8161, LENCEWICZR@GW1.HANSCOM.AF.MIL, 617-377-9369.

The **Center for Software Reuse Operations** (**CSRO**) is an element of the DoD Software Reuse Initiative under the Defense Information Systems Agency/Center for Information Management (DISA/CIM). CSRO is responsible for the Defense Software Repository System (DSRS), an automated library of Reusable Software Components (RSCs) available to the DoD and other government agencies, including supporting contractors. RSCs are products such as requirements, design specifications, architectures, design diagrams, source code, documentation, and test suites.

CONTACT: Customer Assistance Office CSRO, 500 North Washington Street, Suite 101, Falls Church, VA 22046; Fran Becker (703) 536-7485, FAX (703) 536-5640.

COSMIC is NASA's Computer Software Management and Information Center, the central repository for software developed under NASA funding. Programs are made available for reuse by domestic industries, government agencies, and universities under NASA's Technology Utilization Program. New programs are continually added to the repository. In addition to a catalog, released in January of each year, COSMIC offers the following services at no charge:

- a nine month subscription to its quarterly publication, Cosmic Update,
- a custom search of its inventory based on key words,
- a monthly email listing of new and updated programs sent to an Internet address,
- Email user group conferences, via Internet, for the computer programs: CLIPS, TAE and NQS. Components developed under NASA's Technology Utilization Program include computer graphics, structural mechanics, aerodynamics, thermal analysis, control systems design, trajectory determination, CAD/CAM, artificial intelligence, expert systems, and linear algebra subprograms. Many are oriented for avionics applications. Source code is provided with purchase. You can also market your developed software through COSMIC.

  CONTACT: COSMIC, University of Georgia 382 Broad Street Athens, GA 30602, Patricia Mortenson (706) 542-3265, FAX (706) 542-4807, Internet: service@cossack.cosmic.uga.edu, Telnet: cosline.cosmic.uga.edu, (706) 542-7354 (1200/2400 baud, 8 bits, no parity, 1 stop).

GKS/Ada contains ANSI Standard GKS Level 2b of interactive graphics programs. Available for 286/386/486 PC's, DEC VAX hosts, interactive plus SCO Unix and X-Windows, Alsys and Dec compilers, drivers such as VGA, EGA, CGA, Tektronik, hard copy devices, and Metafiles. It features output primitives, segmentation, interactive input, and windows. Size is approximately 60K lines of Ada statements. Source code is available. Prices range from $795 for 286 DOS to $3995 for DEC minicomputers, quantity and educational discounts. **CONTACT**: Software Technology Inc., 1511 Park Avenue Melbourne, FL 32901, Greg Saunders (407) 723-3999, FAX (407) 676-4510.

GRACE (Generic Reusable Ada Components for Engineering) is a library of 275 Ada software components based on commonly used data structures such as strings, matrices, lists, stacks, queues, trees, and graphs. Each component includes complete requirements specifications, design documentation, source code, and one or two test programs. GRACE is completely portable and fully warranted. **CONTACT**: EVB Software Engineering, Inc. 5303 Spectrum Drive Frederick, MD 21701, Jennifer Lott (will supply a free sample) (301) 695-6960 (800) 877-1815 FAX (301) 695-7734 grace@evb.com.

HALO Professional consists of a library of graphics subroutines for MS DOS/Alsys and Meridian, with more than 200 software routines for graphic application development. **CONTACT**: Media Cybernetics, Inc. 8484 Georgia Avenue Silver Springs, Maryland 20910, Doug Paxson (301) 495-3305 X218 FAX (301) 495-5964.

Math Pack contains over 350 Ada mathematical subprograms in 20 reusable generic Ada packages. It includes linear algebra, linear system solutions, integration, differential equations, eigensystems, interpolation, probability functions, Fourier transforms and transcendental functions, and a package called TRANSFORM_PAC containing various FFTs and amplitude determination algorithms for digital signal processing. In the process of release is Stat Pack, a library on numerical statistical properties and functions that will include regression analysis, ANOVA analysis, nonparametric tests and measures, hypothesis testing and analysis, probability distributions, random number generation, and input and output programs for statistical analysis. Purchase includes source code, a comprehensive User's Guide, on-line help and telephone support. **CONTACT**: MassTech, Inc., 3108 Hillsboro Drive Huntsville, AL 35805, Mark Thompson, (205) 539-8360 FAX (205) 533-6730.

MCC is a research consortium involved in "pre-competitive research." It takes technology through the research prototype stage and then a product is developed in any number of ways - a company participating in (paying for) the research incorporates the technology in a product or possibly in an internal process. MCC's Carnot Project is addressing the problem of logically unifying physically distributed, enterprise-wide heterogeneous information, in a system aiding the extraction of reusable assets. MCC publishes Collaborations, a quarterly newsletter reporting on the activities of this research consortium. **CONTACT**: Microelectronics and Computer Technology Corp., 3500 West Balcones Center Drive, Austin, Texas 78759-6509, (512) 343-0978, Cynthia Williams, (512) 338 3512 williams@mcc.com.

NAG Ada Library is a library of reusable mathematical components including solvers for ODE's, PDE's, Curve and Surface fits, extended arithmetic, and others. **CONTACT**: Numerical Algorithms Group, Inc., Sales Department, 1400 Opus Place, Suite 200, Downers Grove, IL 60515-5702; John Zurawsi (708) 971-2337 FAX (708) 971-2706.

N.A. Software Ada Encyclopedia is a wide ranging numeric library providing a toolkit of mathematical utilities for engineers and scientists who write numerical software in Ada. Currently the Encyclopedia contains volumes on:

- Basic facilities: complex arithmetic, elementary functions, vector and matrix arithmetic, and input/output for libraries types,
- Statistics: mean, standard deviation, etc., for grouped and ungrouped data sets, regression analysis, and statistical distribution functions,
- Signal processing: large number of facilities for manipulating and computing with vectors and matrices of value in signal and image processing,
- Fixed-point facilities: a full set of elementary functions,

- Variable precision floating-point: all basic arithmetic operations and elementary functions for arbitrarily high precision,
- Special functions (i.e., Bessel functions and Airy functions),
- High level linear algebra: specialist high level routines covering solution of sets of linear equations, eigenvalue and eigenvector calculations, and linear least squares. A wide range of implementations is available.

  CONTACT: NA Software Limited Merseyside Innovation Centre, 131, Mount Pleasant, Liverpool L3 5TF, United Kingdom, Yvette Gray, +44-(0)51-709 4738 FAX +44-(0)51-709 5645.

Navy Wide Reuse Center (NWRC) provides a comprehensive reuse support environment for all Navy reusable components and provides interfaces to other DoD and non-DoD repositories as well as information on commercially available reusable components. For access, send a request to the program manager. **CONTACT**: NWRC, Washington Navy Yard Building, 196 Code N53, Room 4508, Washington, DC 20374; Angus Faust, Project Manager (202) 433-0718.

NTIS (National Technical Information Service), a self supporting agency of the U.S. Department of Commerce, provides a free products and services catalog. It markets about 5000 microcomputer and mainframe programs and data files and a collection of technical reports containing thousands of computer-related documents. **CONTACT**: National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161, Sales (703) 487-4650 (800) 553-NTIS, pfeinste@DGIS.DTIC.DLA.MIL.

PragmAda Software Engineering offers a variety of low-cost, high quality reusable Ada components. The company guarantees the correctness of all components. Semi-custom development of reusable Ada components is also available, as well as custom Ada development. **CONTACT**: PragmAda Software Engineering, P.O. Box 1533, Sterling, VA 20167-8455, Jeffrey R. Carter (703) 406-3527.

Proplink is an Ada program for analysis of communication link propagation paths from ELF to EHF frequencies. **CONTACT**: IWG Corporation, 1940 Fifth Avenue, Suite 200, San Diego, CA 92101, Larry Gratt (619) 531-0092.

RAPID (Reusable Ada Products for Information System Development) publishes guidelines and standards: ISEC Reusability Guidelines (12/85), ISEC Portability Guidelines (12/85), and RAPID Center Standards for Reusable Software (12/89), available free upon request. RAPID Center Library is an automated catalog and retrieval system allowing a user to identify and extract Reusable Software Components (RSC) meeting specific functional requirements. A faceted classification scheme allows retrieval of lists of relevant RSCs, together with information about the number of times each RSC was extracted, number and description of reported problems, reusability and complexity measures, RSC abstract, and supporting documents. Free information to government contractors. **CONTACT**: RSC Team, RAPID Center USAISSDCW, ATTN: ASQB-IWS-R STOP H-4, Ft. Belvoir, VA 22060 - 5456, Donna Williamson, (703) 285-6272, AUTOVON 356-6202 WRCK@MELPAR-EMH1.ARMY.MIL.

Rockwell International Corporation maintains a database of tools, software components, data, etc., accessible to all Rockwell software engineers. **CONTACT**: MGR, Software Engineering Process Group, M/S 460-225 3200, E. Renner Road, Richardson, TX 75081-6209, J.F. Frizzell (214) 705-0000 ext 3635.

Software Productivity Consortium (SPC) is a jointly funded research and development effort of 12 major aerospace and defense companies (Boeing, General Dynamics, Grunman, Harris, Hughes Aircraft, Lockheed, Loral, Martin Marietta, Northrop, Rockwell, United Technologies, and Vitro). It provides a reuse library tool with facilities for storing, classifying and retrieving reusable components in the form of code, specifications, design representation, test cases, etc. The library supports both hierarchical and faceted classification schemes within either centralized or distributed libraries. The library tool is free to SPC member companies and their subcontractors, and to government sites receiving the members' products. **CONTACT**: Software Productivity Consortium, 2212 Rock Hill Road Herndon, VA 22070, Ron Damer (703) 742-8877 FAX (703) 742-7200.

STARS (Software Technology for Adaptable, Reliable Systems) Program has developed OS components, DBMS, user interface, network communications, text processing, graphics, stream data types, etc. See ASSET listing above.

STO (Software Translation and Optimization) maintains a database of over 15,000 government produced programs in a variety of languages. It tracks all software coming out of the government locating and customizing reusable programs, including mathematical routines, software tools, radar software, neural nets, etc. Consultations available on the management and economic aspects of reuse. STO publishes an annual directory of government software. Cost of the 1992/1993 edition is $149. **CONTACT**: Source Translation & Optimization, P.O. Box 404, Belmont, MA 02178, Greg Aharonian: (617) 489-3727 srctran@world.std.com.

F.3 Reuse Information

The **Ada Directory** is a complete international Ada language product directory. It provides managers, software tool specialists, and software developers, at a cost, with a complete, up-to-date identification and description of:

- Ada tools (including validated compilers).
- Ada development and target environments.
- Reusable components.
- Addresses and telephone numbers for all vendor contracts.
- An extensive index giving vendor company names, product names, and product categories.

- Ada information resources.

  CONTACT: DG Innovations, 8137 Sagimore Court, Fort Wayne, IN 46835 Dale J. Gaumer (219) 485-8952, FAX (219) 429-8445, C566DJG%C2.Mageo.Com@uunet.uu.net.

Ada Information Bulletin Board is a publicly available source of information on the Ada language and Ada activities. The public subdirectory contains Ada material available for downloading. **CONTACT**: Commercial: (703) 614-0215 (300-2400 baud, 8, N, 1) Autovon 224-0215, ftp ajpo.sei.cmu.edu, login: anonymous, password: guest, cd public (enabling dir, get, etc.).

The **Ada Information Clearinghouse** publishes two flyers about Ada and reuse. One flyer provides descriptions of available reusable components and how to obtain them; the other provides a current bibliography of articles related to Ada and reuse. **CONTACT**: Ada Information Clearinghouse, c/o IIT Research Institute, 4600 Forbes Blvd, Lanham, Maryland 20706-4320, Susan Carlson, (703) 685-1477, (800) AdaIC-11, FAX (703) 685-7019, adainfo@ajpo.sei.cmu.edu.

ASR (Ada Software Repository): see products in Section F.2.

The **CARDS (Central Archive for Reusable Defense Software)** Program is a concerted DoD effort to transition advances in the techniques and technologies of library-assisted, domain-specific software reuse into mainstream DoD software procurements. To assist the DoD in identifying and overcoming obstacles to domain-specific reuse, CARDS is developing a Franchise Plan describing, in precise steps, a scenario for implementing a domain-specific library. In addition, CARDS is developing supporting training plans and packages as well as handbooks for acquisition executives (*Direction Level Handbook*), program managers and government contracting personnel (*Acquisition Handbook*), system/software engineers (*Engineer's Handbook*) and component and tool developers (*Component Provider's and Tool Developer's Handbook*). Currently the public can not access CARDS, all users must be working on a government contract. This may be opened up in the future. Most of the CARDS documents are available to the public. **CONTACT**: Robert Lencewicz, ESC/ENS Hanscom AFB, MA 01731-2816, (617) 377-9369, DSN 478-9369, Internet: lencewiczr@gw1.hanscom.af.mil.

NTIS (National Technical Information Service): see products in Section F.2.

ReNews is an electronic software newsletter, open to the public, that includes lists of current and forthcoming reuse conferences, workshops and various papers on reuse experiences and research. In 1992, the ReNews electronic newsletter became affiliated with the IEEE-CS Technical Committee on Software Engineering (TCSE), forming the Subcommittee on Software Reuse. Its membership includes the registered subscribers of ReNews and others interested in advancing software reusability and related technologies. Mailing lists for ReNews and the subcommittee are maintained by TCSE. Reuse subcommittee members automatically become members of the TCSE, which publishes a newsletter three times a year emphasizing current events, awareness, and technology transfer in the diverse software engineering community. The Reuse subcommittee and TCSE sponsor and support various conferences, meetings and activities in the software reuse field. TCSE has five subcommittees in special-interest topic areas: Software Engineering Standards (SES), Reliability Engineering Reverse Engineering Software Reuse (** ReNews **), and Quantitative Methods, At this time, TCSE does not charge any membership dues. However, under IEEE-CS rules, only CS members can receive postal newsletters at no charge. **CONTACT**: Bill Frakes, Virginia Tech., 400 Drew Court Sterling, VA 22170, frakes@neptune.cs.Virginia.edu, 703-698-7020, e.chikofsky@compmail.com, 617-272-0049.

RIG (Reuse Library Interoperability Group) is drafting standards for the interoperability of reuse libraries in areas such as nomenclature, interchange protocols, and software component exchange formats. It is a volunteer, consensus-based organization composed of members from government, academia, and private industry. **CONTACT**: rig@stars.ballston.paramax.com, RIG Secretariat c/o Applied Expertise, 1925 N. Lynn St., Suite 802, Arlington, VA 22209, (703) 516-0911.

DISA and RAPID have prepared a set of reuse guidelines:

- *ISEC Portability Guidelines discusses the issues and tradeoffs that must be made in writing a portable Ada program. Discussions center on three aspects of portability: source modification, program behavior, and safety.

- ISEC Reusability Guidelines addresses the design and development of reusable software in Ada, beginning with the concept of software reusability and issues to be considered; specific guidelines for design, and interfaces; and documentation and management.

- JAMPS PDL Guide defines a program design language used to provide a notation for expressing and manipulating the system design of the JINTACCS Automated Message Preparation System.

- ISEC Ada Investigation presents a comprehensive set of practices for the systematic development and maintenance of large systems in Information Systems Engineering Commands Management Systems.

- CIM Reuse Program Design/Coding Guidelines for Reusable Ada Software is used by DoD Software Warehouse staff to evaluate for quality and completeness, the code, format, style, and documentation of reusable software components submitted to the Software Warehouse.

- RAPID Center RSC Procedures sets forth procedures used by the DoD Software Warehouse staff to identify, evaluate, prepare, install, and maintain reusable software components, with a "cradle-to-grave" life-cycle view of the reusable software component process. *System Administrator's Guide is used by DoD Software Warehouse staff as guidance for the effective use of the DoD software reuse tool to accomplish system administration tasks.

CONTACT: See the previously mentioned RAPID and DISA contacts.

Software Technology Support Center (STSC) has been designated as the Air Force Focal Point for Software Technology. This includes:

- The collection, exchange and distribution of software technology information, evaluation of existing and new technologies.
- Consulting with individual Air Force Software Development and Support Activities (SDSAs) and the corresponding insertion of software technologies to improve the development processes.

STSC is working in coordination with the branches of government, academia, and industry to garner information from organizations involved in software engineering and apply it in a consulting role. Additional activities include classification of information in various software technology domains and assistance in performing evaluations. Specifically, the software reuse technology domain will be investigated with the prime aim of assisting SDSAs in process improvement as it relates to reuse. For instance, information sources such as software periodicals, proceedings, textbooks, seminars, conferences, etc., will be explored according to categories of methodologies, programming environments, models (including domain analysis), reuse programming design techniques, and management/non-management issues. Furthermore, inquiries into guidelines for coding styles, documentation formats, testing requirements, cataloging techniques, tools providing/promoting reuse functions, and development methodologies supporting reuse in embedded software will be made and the resulting analyses will be provided. The intention is to identify possible application domains where software reuse may be feasible and assist clients in the related implementation and insertion. STSC will determine the commonalities and differences between these application domains as they affect software reuse requirements, and describe the most effective tools, methods, and environments for each of the described domains, including an assessment as to the needs that are not currently met. STSC provides a free subscription to "Crosstalk", the Journal of Defense Software Engineering and a Bulletin Board System on (801) 774-6509 or Telnet 137.241.33.1. **CONTACT**: Software Technology Support Center, OO-ALC/TISE, Hill AFB, UT 84056, Technology: Gary Peterson, (801) 777-4952, peterseg@oodis01.hill.af.mil; Reuse: Larry Smith, (801) 777-6424, bbliss@oodis01.hill.af.mil; BBS: George A. Klipper, (801) 777-9712, klipper@oodis01.hill.af.mil; FAX (801) 777-8069.

Technology Transfer International, Inc. has published "Software Reuse in Japan", a comprehensive 360 page report dealing with issues such as how to build software systems by combining existing components, modified code and new code, management practices in Japanese software factories, technologies and infrastructure supporting software reuse, government research and development programs in software engineering, software metrics for reuse and quality, barriers to implementation and how to handle them, and the industrialization of the software engineering process. They have developed a lecture series on software factories including the Hitachi factory and tools for National Technical University and plan to market tapes and lecture notes. **CONTACT**: Technology Transfer International, Inc., 6736 War Eagle Place, Colorado Spring, CO 80919-1634; John Morrison, (719) 260-0925, FAX: (719) 593-2430, 70712.3561@compuserve.com.

APPENDIX G - RELATED DOCUMENTS

G.1 CARDS Products and Services

The Central Archive for Reusable Defense Software (CARDS) Program has produced a significant collection of documents detailing its research efforts in the area of domain-specific reuse. In addition, CARDS sponsors seminars and workshops, and is implementing a market study to assess DoD's current reuse state-of-practice processes. To propagate the information contained in the documents and the information obtained through seminars, workshops, and studies, CARDS has developed a training effort for which it provides both documentation and actual course presentations. Some CARDS documents detail the CARDS experience in developing and supporting a domain-specific reuse library. These documents can be used as models by organizations interested in implementing their own domain-specific library. Copies of any of the CARDS documents can be optained by calling the CARDS Hotline at (800) 828–8161.

CARDS has identified four significant goals to begin the transition of domain-specific software reuse techniques and technologies into DoD software procurements. Following is a list of these four goals together with a description of the related CARDS products and services:

1. **Goal 1**: Produce, document, and propagate techniques to enable domain-specific reuse throughout DoD.

   - The *Acquisition Handbook* [CARDSa] is aimed towards government program managers and their support personnel, such as contracting officers and administrators, involved in systems, subsystems, and component acquisition. The concepts discussed assume the reader has several years experience in acquisition. The handbook assist readers in incorporating software reuse into all phases of the acquisition lifecycle. It is meant to help develop and tailor reuse programs. Software reuse methods, examples, recommendations, and techniques are presented. Implications and affects of software reuse on technical, management, cost, schedule, and risk aspects of a program/system during the acquisition process are provided.

   - The *Direction Level Handbook* [CARDSb] is directed towards acquisition executives of the services to facilitate the institutionalization of software reuse. The audience of Program Executive Officers (PEOs), Designated Acquisition Commanders (DACs), and their supporting staff, are provided with a framework to assist in establishing plans to manage reuse across their systems and to reach the goals outlined in the DoD Software Reuse Vision and Strategy [?]. This handbook assist in incorporating software reuse into the initial planning stages of an acquisition, as well as at critical points within the acquisition lifecycle.

- The *Engineer's Handbook* [CARDSc] provides guidance to government System Program Office (SPO) engineers on envisioned changes to their duties and responsibilities as domain-specific software reuse becomes incorporated into mainstream DoD system/software acquisition and engineering processes. The intended audience is SPO engineers who are responsible for pre-Request For Proposal (RFP) engineering activities, proposal evaluation, monitoring of engineering activities after a contract is awarded, and monitoring of ongoing sustaining (or maintenance) engineering efforts of fielded products. To fully utilize the concepts in this handbook, it is recommended the reader be familiar with software development techniques and methodologies, existing government regulations, and the acquisition process.

- The *Component Provider's and Tool Developer's Handbook* [CARDSp] provides government software developers and industry vendors (e.g., government contractors and commercial software creators) with guidance for building domain-specific reusable components and tools to facilitate software reuse. The goal is to stimulate the development and commercialization of large-scare reusable components and tools and to complement the CARDS Engineer's Handbook. The end users of these components and tools are system developers on government programs. The main audience is government domain engineers, and component and tool creators (either government or contractors). SPO engineers should be familiar with this handbook.

- The *Training Plan* [CARDSj] is aimed towards DoD and DoD industry personnel, undergraduate and graduate computer science students, and software engineers. It recommends how to conduct top quality training, identifies the necessary functions to support the recommendations, and identifies content methods.

- The *Introduction to Software Reuse Course* provides government, industry, and university students with an understanding of reuse and domain-specific reuse. Domain engineering is defined and the domain analysis products are explored. Reuse library concepts are discussed and an actual library is demonstrated.

- The *CARDS Program and Library Course* introduces individuals and organizations to CARDS and the CARDS Command Center Library (CCL). It explains the major goals of CARDS, its products and services, and provides detailed guidance on how to utilize the CARDS CCL. It is assumed the student has a solid understanding of basic reuse concepts, but little or no knowledge of CARDS.

- The *Software Architecture Seminar Report* [CARDSl] is based on a CARDS seminar and workshop to increase awareness, explore current research into software architectures as a means of implementing software reuse, and examine current practices and issues involving architectures. This report includes: presentation slides, panel discussions, the workshop, questions and answers, and issues discussed.

2. **Goal 2**: Develop a Franchise Plan which provides a blueprint for institutionalizing process-driven, domain-specific, architecture-centric, library-assisted software reuse throughout DoD.

    - The *Franchise Plan* [CARDSk] references a series of handbooks, library documentation, and training materials, each targeted to a particular audience which needs to be involved in making reuse happen in DoD software procurements. The purpose of this plan is to assist with the implementation of a process-driven, domain-specific, architecture-centric, library-assisted software reuse infrastructure with DoD organizations. The plan defines a planning process necessary to build a reuse infrastructure. It is a tool management can use to develop a detailed, tailored implementation plan to prepare an organization to begin the software reuse process.

3. **Goal 3**: Implement the Franchise Plan with selected organizations and/or provide a tailored set of services to support reuse.

    - CARDS provides interested parties with technical expertise in the field of process-driven, domain-specific, architecture-centric, library-assisted software reuse. CARDS consulting services include reuse adoption support, domain analysis/modeling, complete site and operation survey, feasibility studies, and operational hardware and software installation. CARDS will:

        - Provide domain engineers with an implementation framework for reuse libraries in their domain of interest.

        - Guide application engineers through the selection of alternative components based on generic architectures.

4. **Goal 4**: Develop and operate a domain-specific library system and necessary tools.

    - A library model (based on the Software Technology for Adaptable, Reliable Systems (STARS) Reuse Library Framework (RLF) technology) has been created and Portable, Reusable, Integrated Software Modules (PRISM) Program components are being added to the library. The library provides a view of command center requirements presented in the Defense Information Systems Agency (DISA) Command Center Design Handbook [DISA91] and provides a mapping of those requirements to the Generic Command Center Architecture (GCCA). The CARDS CCL demonstrates the potential of library-assisted, domain-specific reusability based upon domain-specific software architectures. Besides the CCL, CARDS has a second library consisting of available PRISM documents. A third library consisting of CARDS documents is being created.

- The *Technical Concepts Document* [CARDSd] baselines the technical foundation for the CARDS CCL and for other domains. It describes the reuse library infrastructure as it pertains to the CARDS CCL. Basic reuse concepts are discussed, such as the relationship of domain engineering to system engineering, domain modeling concepts, operational concepts, and tools used. Technical aspects necessary to make the CARDS CCL operational and key technical interoperability and security concepts are addressed.

- The *Library Development Handbook* [CARDSe] provides an overview of the phases involved in developing a domain-specific reuse library: domain analysis, library encoding, and library population. This handbook presents a CARDS generic library population process. The handbook enumerates specific instructions and examples for populating a domain-specific reuse library, based on this library population process.

- The *Library Model Document* [CARDSn] describes the current state of each CARDS library, e.g., CCL. It is updated with every library release. It describes how software engineering relates and feeds back to domain engineering, how domain engineering compares and contrasts to library modeling, and examines modeling concepts and specialization/aggregation hierarchies. The intended audience is anyone desiring an understanding of one or more CARDS library and wanting a view of the current library release. A knowledge of software engineering concepts is assumed.

- The *Library User's Guide* [CARDSq] describes how to access the CARDS libraries. It describes some of the CARDS specific aspects of RLF. An elementary explanation of RLF concepts is provided.

- The *Version Description Document* [CARDSr] provides information about each release, including release notes (changes from the previous version and possible problems and known errors), installation instructions, and a listing of computer media files.

- The *Library Operations Policies and Procedures* [CARDSf and g] documents the policies and procedures required to implement and maintain a reuse library. It addresses the information concerning strategies for managing and maintaining an existing reuse library system. A high level (non-CARDS specific) view of the policies and procedures for library operations is given for upper level management and technical supervisors (Volume I). Day-to-day CARDS operating instructions needed by technical personnel are also detailed (Volume II). In addition, the processes for developing the contents of a domain-specific library are specified.

G.2 Non-CARDS Products

The following documents were not prepared by CARDS, but are related to the CARDS software reuse effort:

- The Software Technology for Adaptable, Reliable Software (STARS) Program's *Conceptual Framework for Reuse Processes* (CFRP) defines a context for considering reuse-related software development processes, their interrelationships, and their composition and integration with each other and with non-reuse-related processes to form reuse-oriented life cycle process models [STARS92].
- The STARS *Reusability Guidelines* proposes guidelines for the design and coding of reusable software components. It offers common capabilities within well-defined application domains that are suitable for installation in a library of reusable components. It is directed to software developers and maintainers of a software reuse library [STARS89].
- *SDP-2000: A Guide To Project Implementation of Megaprogramming* describes a vision of software industry as it may exist under megaprogramming (when superior practices in software engineering are synthesized) and describes transition steps to be required by the government and contractors alike [POORE93].
- *A New Process for Acquiring Software Architecture* outlines a process used to ensure system acquisitions include attention to the software architecture [SAUNDERS93].